University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Multigrain Parallelism: Compiling Coarse-Grain Parallel Programs for Fine-Grain Execution

Jaime Arteaga, Stephane Zuckerman, and Guang Gao

CAPSL Technical Memo 130
April, 2016

Contents

Abstract

The overwhelming wealth of parallelism exposed by Extreme-scale computing is rekindling the interest for fine-grain multithreading. Indeed, popular parallel programming models, such as OpenMP, are integrating fine-grain tasking in their newest standards. Yet, classical coarse-grain constructs are still largely preferred, as they are seen as simpler to express parallelism. In this document, we present a multigrain parallel programming environment that allows programmers to use these well-known coarse-grain constructs to generate a fine-grain multithreaded application, which runs on top of a fine-grain event-driven program execution model. Experimental results with two scientific benchmarks show that fine-grain applications generated by our environment are highly competitive or even outperform their OpenMP counterparts: we reach speedups ranging from 0.9× to 1.1× for a molecular dynamics simulation (NWChem-SCF) and up to 1.7× for a graph benchmark (Graph500).

1 Introduction

In order to reach Exascale performance, several challenges must be overcome in terms of core per chip count, power, system memory, data movement, and resiliency [1]. This has resulted in significant changes in hardware over the past few years. More cores are being packed on a single chip, more complex interconnections and memory hierarchies are being designed, and the use of accelerators has become ubiquitous on almost every supercomputer. Furthermore, concerns about energy consumption and fault tolerance are now as important as the speed and the peak performance of any machine.

These changes in hardware must be accompanied by changes in the entire software stack of future supercomputers. In particular, changes must be made to the programming methodologies used to write parallel applications [1–3]. This has renewed the interest on fine-grain programming models in order to handle the high-core count of future Exascale architectures. As a result, the latest versions of legacy application programming interfaces (API), such as OpenMP, have been updated with fine-grain constructs like `task`. However, classical coarse-grain constructs are still used and present on countless applications, reducing the ability to take full advantage of the high-node concurrency found in current and future architectures.

APIs for fine-grain program execution models (PXM), on the other hand, are better for these types of architectures by providing programmers with a more complete interface to decompose an application into a great number of small tasks that can be easily scheduled and moved among cores.

A fine-grain PXM with such APIs is the *Codelet model* [4]. The Codelet model is a fine-grain event-driven PXM that provides programmers with a comprehensive interface to develop parallel programs with several levels of granularity and parallelism and to interact with the highly heterogeneous parallel hardware foreseen on Exascale computing. Some of the runtimes implementing the Codelet model are *Delaware Adaptive Runtime System (DARTS)* [5], *Open Community Runtime (OCR)* [6], and *Swift Adaptative Runtime Machine (SWARM)* [7]. Even though these APIs are more suitable to exploit the increasing number of cores in a shared-memory node, scientific community and main users of high-performance systems still heavily rely on OpenMP given its popularity and familiarity with it. In addition, porting an existing OpenMP application to fit a new PXM, such as the Codelet model, can be time consuming and error prone.

As a solution, we present a multigrain parallel programming environment, which takes programs parallelized with coarse-grain constructs (like those found in OpenMP) and generates an equivalent fine-grain event-driven application that fully exploits the high node concurrency available in the underlying hardware. By using this environment, programmers will be able to keep using OpenMP applications and, at the same time, benefit of the advantages offered by a model specifically designed for fine-grain execution, such as the Codelet model, allowing a gentle transition into more explicit fine-grain programming models over time.

The major component in our environment is *omp2cd*, a multigrain parallel compiler based on *clang* [8], which transforms an OpenMP input program into a fine-grain application following the Codelet Model. During this transformation, the compiler automatically determines the granularity of the event-driven tasks at several levels of parallelism depending on the type of parallelism present in the application and the OpenMP directives used by the programmer. Once generated, the output application is executed on a customized version of DARTS that complements the optimizations made at compile time by *omp2cd*.

This paper presents the following contributions:

- *omp2cd*, a multigrain parallel compiler to port OpenMP applications into the Codelet Model, a fine-grain event-driven program execution model for architectures with high-node concurrency.
- An improved version of DARTS that reduces the runtime's overhead and complements the optimizations performed at compile time by *omp2cd*.
- Experimental evaluation of the multigrain parallel programming environment using two scientific benchmarks: Graph500 and NWChem-SCF, showing the fine-grain applications generated by *omp2cd* are highly competitive and can outperform their OpenMP counterparts.

This paper is outlined as follows: Section 2 presents a background in the Codelet model and DARTS, followed by Section 3, where the *omp2cd* compiler, the major component of our multigrain parallel programming environment, is discussed. Section 4 explains some of the optimizations introduced in DARTS to complement the compile-time optimizations, while Section 5 presents the experimental evaluation of the environment using Graph500 and NWChem-SCF. Finally, Section 6 discusses some of the related work and Section 7 presents the major conclusions derived from this work and outlines the future work.

2 Background

One of the fine-grain PXMs that have been designed as a response to the challenges faced in Exascale computing is the Codelet model. The fine-grain tasks generated in our environment follow the specifications of this model.

2.1 The Codelet Model

The Codelet model is a fine-grain PXM designed to fully exploit architectures with nodes featuring a high degree of parallelism by decomposing an application into a great number of light-weight asynchronous tasks called *codelets*. These tasks can be easily scheduled among cores, depending on resource availability and other constraints, such as proximity to data to compute, power budget, and fault tolerance [4].

Each codelet has a collection of data inputs (and outputs) coming from (and going to respectively) other codelets. Additionally, a codelet may require a set of specific resources for its execution. When both data and resource dependencies are met, a codelet can *fire* (*i.e.* it executes its *fire* method), and be scheduled atomically for execution. A codelet runs until completion (*i.e.*, a codelet is *non-preemptive*).

Codelets are grouped into asynchronous functions called *Threaded Procedures* (TPs). A TP is assigned to a specific group of resources, *e.g.*, a group of cores and some memory. Codelets belonging to the same TP also share its *frame*, *i.e.*, they have access to both the data and code it holds. As a result, grouping codelets to share the same data set allows for improved locality. All data used by the codelets of the same TP, including inputs, outputs, and local variables, are contained in the TP's frame. A TP can be invoked at any time during the lifetime of the application and exists until all its active codelets have met their dependencies and then fired.

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

2.2 DARTS

To execute the fine-grain applications generated by our environment, we use *DARTS* (Delaware Adaptive RunTime System), an implementation of the Codelet Model written in C++ [5]. We selected this runtime because it provides a complete interface to create codelets and TPs, allowing having applications with at least two levels of parallelism, and because it is publicly available from [9].

In DARTS, TPs and codelets are scheduled using a two-level scheduling mechanism composed of *TP* and *Micro* schedulers. When a TP is invoked, a handle for it is created and assigned to a TP scheduler, which initializes the TP and instantiates all of its codelets. Then, when one of these codelets is ready to be fired, the TP scheduler assigns it to one of its Micro schedulers for execution. In the case where all the Micro schedulers are busy executing other codelets and cannot accept additional ready codelets on their queues, the TP scheduler may opt to execute a ready codelet itself. If there are no codelets to execute (or TPs to be initialized in the case of TP schedulers), the schedulers will spin until they have work to perform.

Users can specify the number of TP and Micro schedulers to be used in an application upon instantiation of the runtime, as well as their scheduling policies. The two main types of policies are *static* and *dynamic*. When using a static policy, a TP scheduler only initializes TPs assigned locally by one of its Micro schedulers, while with dynamic, it may also steal TP handles from other scheduler's queue.

A Micro scheduler with a static policy only executes codelets assigned by its TP Scheduler. However, if its policy is dynamic, the Micro scheduler may steal codelets from the TP scheduler's ready-codelet queue.

Finally, users can also select the affinity scheme used to pin down the schedulers to the cores, either in a compact or spread manner.

3 Multigrain Parallel Compiler omp2cd

To generate the fine-grain tasks defined in the Codelet model from an OpenMP application, we designed the *omp2cd* multigrain parallel compiler. Along with an improved version of DARTS, this compiler provides a complete multigrain parallel programming environment for the execution of coarse-grain applications in a fine-grain programming execution model.

3.1 Compiler Overview

A diagram of the compiler's internal structure and its interaction with other components in the environment is presented in Figure 1. The compiler was designed using the tools available in clang 3.5 [8]. Currently, *omp2cd* supports the C language, implements the most-used directives, constructs, and runtime library routines from OpenMP v2.5 [10], as well as the *task* and *taskwait* constructs defined in v3.1 [11].

The compiler receives a set of OpenMP source files and uses clang's *ASTFrontendAction*s to recursively access their ASTs and to construct an annotated AST for each input file. These annotations are used to collect all the necessary information to later optimize the input application for a finer-grain execution using the Codelet model. Some of the annotations made to the nodes in the clang's ASTs are:

- *Node's ID:* Unique identification number for each AST node across all translation units.
- *Inputs Vector:* If the node represents an `omp` region, this vector stores references to the variables declared outside the region but used inside of it. Each element in the vector also stores the input's type, its OpenMP data-sharing attributes inside the region (*e.g.* `shared`, `private`, or `firstprivate`), and a pointer to the node with the variable's declaration, among other information.
- *Variables Vector:* If the node represents an `omp` region, this vector stores references to the variables declared and used inside of it. In the case of a function declaration, the vector also stores information about its arguments. This structure and the inputs vector are especially useful when creating TPs and declaring on its frame the variables used by its codelets.
- *Loop Info:* Stores useful information to create the TPs and codelets associated with a `for` region. Information includes the loop iteration variable, the scheduling policy, and the reduction variables (if any), *etc.*

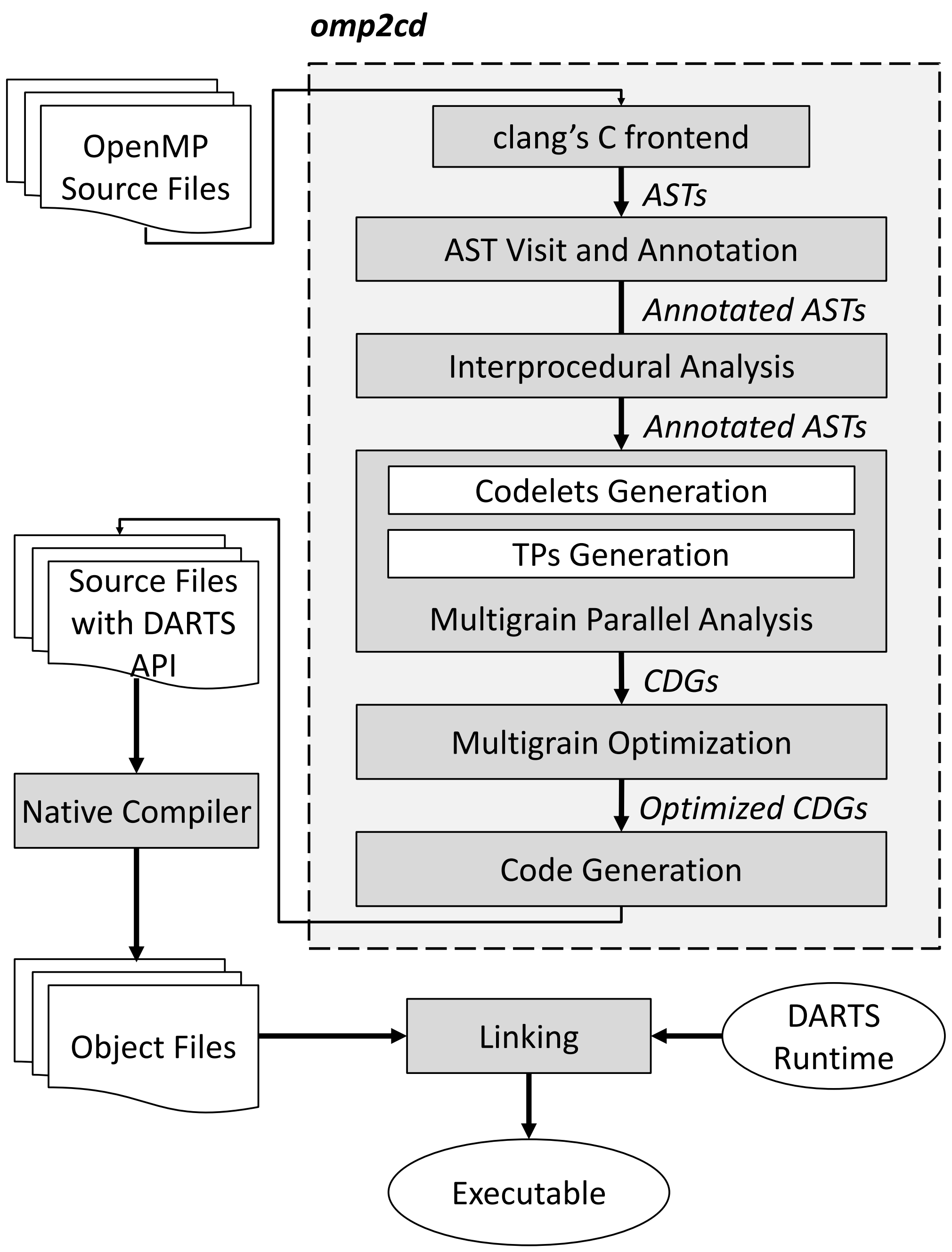


Figure 1: *omp2cd* Internal Structure and Interaction with Other Components of the Environment.

The annotated ASTs then go through an interprocedural analysis to identify the functions containing `omp` executable directives and/or being called from `omp` regions. This is required in order to correctly generate DARTS constructs for these functions and their calls. Special cases are handled like functions with `omp` directives on their bodies but called from both sequential and parallel sections in the application. In this case, the functions are marked to be printed in the output files in both their sequential (original) form and their DARTS version.

After the interprocedural analysis, the annotated ASTs are converted to a collection of *Codelet Graphs* (CDG) by grouping nodes in the ASTs in codelets and arranging codelets in TPs. This is done through a *Multigrain Parallel Analysis*, presented in the following sub-section, that examines the `omp` directives and parallelism available in the application.

The CDGs then go through a *Multigrain Optimization* stage to optimize the size and number of TPs and codelets and improve the application's performance in a fine-grain execution environment. Lastly, the optimized CDGs are translated into C++ code. The output is an application ready to be executed on DARTS.

3.2 Creation of CDGs using a Multigrain Parallel Analysis

To generate the CDGs from the annotated ASTs, *omp2cd* uses a Multigrain Parallel Analysis that follows the specifications of the Codelet model. This analysis is performed based on the information annotated in the ASTs, which provides an insight to the parallelism available in the application and the `omp` directives used by the programmer.

The compiler traverses each annotated AST in depth-first order and creates a TP type[1] for each node containing:

1. The declaration of a function containing an `omp` executable directive[2].
2. An `omp` region[3] other than `task`, `atomic` or `critical`, since the first is converted into a codelet, while the last two are translated into atomic statements and/or protected sections of code.

If a function contains, at least, a `barrier` or `taskwait` directive, such function is converted into a TP to implement the synchronization construct. However, finding a node with either one of these constructs does not constitute the creation of a new TP since these are stand-alone directives with no associated code [11].

Once a TP is defined for a node, its children in the AST are grouped into codelets using the concept of *Codelet Basic Blocks (CBB)*. CBBs are similar to the *basic blocks* used for control flow analysis of programs [12]. But in the context of *omp2cd*, a CBB is defined as a straightline sequence of statements that can be executed *by the same codelet type* and does not branch except at the entry and the exit. After defining a CBB, a new codelet type is created and the block's statements are written in the codelet's fire method.

CBB's leaders are defined according to the following rules:

[1]In this paper, a *type* is defined in the context of Object Oriented Programming.

[2]An `omp` *executable directive* is "An OpenMP directive that... may be placed in an executable context [11]".

[3]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..." [11].

1. The first statement in a function or `omp` region converted into a TP is a leader.
2. The first instruction in a branch is a leader.
3. A call to a function converted into a TP is a leader.
4. A statement following a call to a function converted into a TP is a leader.
5. An `omp` executable directive other than `atomic` or `critical` is a leader, since, as mentioned before, they are not converted into TPs and, therefore, do not represent a branch in the program's control flow.
6. A statement following an `omp` executable directive other than `atomic` or `critical` is a leader.
7. The implicit barrier of an `omp` region is a leader.

Depending on the rule used to defined the leader and on the statements in the block, the codelet created for the associated CBB can be any of the following categories:

- *Computation Codelet:* Contains statements that are neither associated with the invocation of TPs nor synchronization constructs.
- *Invocation Codelet:* Created for CBBs containing the invocation of a TP corresponding to either functions or `omp` regions. An invocation 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.
- *Synchronization Codelet:* Represent a synchronization point in the program, mainly defined by `barrier` and `taskwait` constructs and implicit barriers.

Computation codelets may be instantiated *N* times for concurrent execution, with *N* being the number of threads defined for the current `omp` region and set using either `omp_set_num_threads()` or the environment variable `OMP_NUM_THREADS`, for example. Each instance is assigned an ID from 0 to *N-1*, which is returned upon calling `omp_get_thread_num()`.

Invocation codelets may also be instantiated *N* times, each one attempting to invoke an instance of the corresponding TP type. Depending on the computational load of the TP, the compiler may decide to invoke only one or up to *M* instances of the TP, with $M \leqslant N$. In this case, the available threads are distributed evenly among all the instances. If the TP being invoked corresponds to an `omp` region, the value for *M* depends on the region's executable directive (*e.g.* *M* is 1 for a TP created for a `single` region).

To this end, each invocation codelet contains an atomic operation to test and set the number of instances already invoked. If there are instances yet to be invoked, the codelet proceeds; otherwise, it spins (*i.e.* it resets and signals itself) until the pointer to the assigned instance is available. Once the TP instance has been invoked (or its pointer is available), only the codelet in such instance with the same ID as the invocation codelet is fired. This ensures that only threads that have reached the point in the program where the function call (or the `omp` region) is are allowed to proceed.

Synchronization codelets are used to ensure all threads have reached a specific point in the program's control flow. Consequently, only one instance of each synchronization codelet type is used. The only exception are implicit barriers in `for` regions, where the compiler creates a binary tree with $N - 1$ instances to reduce the overhead of having all threads atomically signaling one single instance of the synchronization codelet type.

3.3 Implementation of Specific OpenMP Regions

For some `omp` constructs defined in the ASTs, the process described in Section 3.2 may be altered in order to optimize the number of TPs and codelets in the created CDG.

3.3.1 Single Threaded Regions

Depending on the total number of statements to be executed inside the region, the compiler may implement a `single` or `master` region as a separate TP or as a single computation codelet in the TP corresponding to the enclosing `parallel` region. If the number of instructions is greater than the limit specified by the compiler (or overwritten on the command line by the user), then the compiler implements the region as a separate TP so it can be scheduled to a different TP scheduler; otherwise it implements it as a computation codelet. This is specially useful when these regions have a `nowait` clause, so the TP scheduler executing the enclosing `parallel` region can continue using all of its Micro schedulers.

For `single` regions, an atomic operation is added to choose the thread executing the region's statements, whereas for `master` regions, a conditional statement is used to select the master thread.

3.3.2 Protected Regions

As mentioned on the rules to create CBB's leaders, `atomic` and `critical` executable directives are not necessarily translated into the definition of a new leader (unless, for example, they are the first statement of a function converted to TP or the statement following a call to such kind of function). Therefore, the statements inside an `atomic` or `critical` region can be part of any CBB. For `atomic` regions, an atomic operation is used when possible, while for `critical` regions a mutex is defined to protect all regions with the same name in the application.

3.3.3 Loops

To balance the work among the threads executing a `for` region, the TP created for it may be instantiated *P* times, executing the loop concurrently on *P* different TP schedulers. The *N* computation codelets executing the loop's body are distributed evenly and in consecutive order among all the instances. With *N* being again the number of threads defined for the region, this means the first instance is assigned the codelets with IDs from 0 to $((N/P)-1)$, the second instance the codelets whose IDs go from (N/P) to

$((N*2/P)-1)$, and so on, until the last instance having the codelets with IDs from $((N*(P-1))/P)$ to $N-1$.

The iteration range is distributed among all the *N* codelets based on the specifications for the loop's schedule policy defined in the OpenMP standard [10].

For `static` scheduling, the iteration is first divided by *P*. Then, the resulting subranges are distributed among all the *P* instances of the TP depending on their codelet's IDs to ensure that the assignment of logical iterations is always the same. Finally, each instance of the TP divides its subrange into its N/P codelets, assigning each chunk according to the codelet's ID.

On the other hand, if the loop scheduling policy is `dynamic`, an atomic operation is used to assign chunks to codelets as they request them. The compiler also supports `guided` scheduling, as well as different chunk sizes as specified in the OpenMP standard.

3.4 Implementation of Branches and Loops Containing Invocations to TPs

As explained before, an invocation codelet may invoke the corresponding TP or reset and signal itself until the pointer to an instance of the TP is available. If the invocation occurs inside a branch or the body of a loop statement, the compiler may restructure the statement and/or create additional codelets to ensure that when the invocation codelet resets itself, no other statements in the branch or the loop's body are executed again.

If the invocation occurs from at least one of the branches of an conditional jump, a computation codelet is created to evaluate the condition, whereas each branch is transformed into codelets based on the rules to define CBB's leaders. At runtime, the codelet evaluating the condition signals the first codelet in the selected branch, while the last codelet in that branch signals the codelet containing the statement that follows the conditional jump.

If the invocation occurs in a loop statement, this is decomposed in a computation codelet initializing the loop variable (in the case of `for` loops), another codelet evaluating the condition (and updating the variable in the case of `for` loops), and one or more codelets executing the body. Signaling between these codelets is correctly arranged depending on the loop being transformed (*i.e.* `while`, `do-while` or `for`).

3.5 Multigrain Optimization

After defining the TPs and codelets for each CDG, the compiler applies a series of optimizations at the two levels of parallelism created during the Multigran Parallel Analysis. These optimizations may add, merge, or delete codelets, as well as merge or delete TPs. Adding new TPs is avoided since invoking a TP has a significant runtime overhead. If a TP is deleted, its codelets may also be deleted or added to another TP.

3.5.1 Region Inlining

The concept of implementing some `omp` regions as computation codelets instead of TPs (as explained in Section 3.3 for `single` and `master` regions) is generalized and applied to other `omp` regions depending on the number of instructions to be executed. If this number is smaller than the limit specified by the compiler (or overwritten by the user in the command line), then the region is merged with the enclosing `parallel` region, and thus all the codelets they contain are moved one level "up." The objective is to avoid the overhead of scheduling and instantiating a new TP. This is especially useful, for example, in `for` regions with a single statement initializing a shared array.

3.5.2 Optimization of `for` Regions

If a `for` region cannot be inlined, then the compiler may optimize it by using a set of special parallel constructs provided by DARTS, named `Loops` [5]. With this optimization, the compiler creates three implementations for the `for` region. Then, at runtime, one these implementations is selected based on the current number of iterations to perform, $Niter$, and two threshold parameters defined by the compiler and tuned by the user, T_0 and T_1:

- If $Niter < T_0$, then all iterations are performed by a single computation codelet. This codelet belongs to the TP representing the enclosing `parallel` region.
- If $T_0 \leqslant Niter < T_1$, then a *Codelet Parallel For (CPF) Loop* is used. This construct invokes P instances of the TP representing the loop body, each one with a single computation codelet. The iteration range is distributed evenly among all the *P* instances.
- If otherwise $T_1 \leqslant Niter$, then a *Parallel For (PF) Loop* is used. This construct invokes Q CPF Loops, each one invoking P instances of the TP representing the loop body and containing a single computation codelet.

T_0, T_1, P, and Q are defined for each `for` region in the input code and may be tuned by the user depending on the number of TP and Micro schedulers set for the runtime, as well as the expected maximum iteration range of the loop, and the number of threads set for the region.

Currently, this optimization only supports loops with a static scheduling policy. Also, since with this optimization the loop may be executed with less than *N* threads, the logical iteration partitioning is not always preserved, as specified in the OpenMP standard. As a result, this optimization is not applied automatically but only enabled from the command line so users can evaluate the improvements introduced in the application when applied. Future versions of the compiler will implement this optimization in compliance with the OpenMP standard in terms of scheduling policies and assignment of logical iterations.

3.5.3 Task and Taskwait Transformation

According to the OpenMP v3.1 standard [11], upon encountering a `task` region, data and code are captured but the task may not be executed right away—it is only guaranteed to have been executed before its synchronization point (*e.g.*, `barrier` or `taskwait`). Accordingly, this optimization module looks for the `task`'s synchronization point and makes sure the computation codelet representing the task is signaling it, instead of signaling the next codelet. Additionally, this module checks if a `task` region is immediately followed by its synchronization point and if that is `taskwait`, in which case both are merged into a single computation codelet.

3.6 Code Generation

The last stage in the compiler takes the CDGs optimized by the Multigrain Optimization stage and generates C++ header and source files for each input OpenMP source file. Each of the statements inside a codelet is printed using customized versions of the methods provided by clang to print statements in the AST. The modifications made to these methods allow printing the variables in the statements as scalars or pointers to the frame of the codelet's TP, depending on the data sharing attributes defined for each variable in the enclosing `omp` region. Whenever possible, local variables declared inside loops and `for` regions are left as local variables in the computation codelets executing the loop's body.

This stage also prints all class declarations and other statements specific to DARTS, in order to generate a valid C++ fine-grain application. Most used OpenMP runtime library routines, such as `omp_get_max_threads()` and `omp_set_num_threads(int)`, are also added to the output files at this stage.

3.7 Example

As an example of the multigrain transformations made by the compiler to an input OpenMP application, Figure 2 presents a section of code and the TPs and codelets created after analyzing its annotated AST. The code contains a `parallel` region with two other regions inside (a `single` and a `for`), preceded by a set of statements to be executed concurrently by all threads.

Following the rules to create TPs and to define the leaders of CBBs presented in Section 3.2, the `parallel` region is converted into a TP, while the statements at its beginning are grouped into a single computation codelet type, which is instantiated N times.

The `single` region is converted into a TP with one computation codelet. However, as mentioned in Section 3.3, a computation codelet may have also been used if the number of statements to execute on this region were less than the limit specified by the compiler or the user. N invocation codelets are also created, one for each thread, to determine atomically the thread executing the instructions in the region (for simplicity, this thread is $N-1$ in Figure 2). This region has no implicit barrier so the compiler does not create a synchronization codelet for it and each thread can continue individually.

Assuming the `for` region cannot be inlined and the user decides not to use the DARTS `Loop` constructs, this region is implemented with *P* instances of its TP, as explained in Section 3.3. The *N* available threads are distributed among all the TP instances, with the iteration chunks assigned depending on the scheduling policy set for the loop (`static` by default). Invocation codelets may reset themselves while they wait for the pointer to their assigned instance to be available. This is illustrated in Figure 2 by having a self-loop edge on each invocation codelet. Because of this, invocation codelets cannot merged with other invocation or computation codelets to avoid executing other invocations or statements more than established.

To implement the implicit barrier in the `for` region, a synchronization codelet is added after it. Since this barrier is immediately followed by the implicit barrier of the `parallel` region, the compiler merges both synchronization points in a single codelet.

4 Improvements Introduced in DARTS

To complement the optimizations made at compile time by *omp2cd*, the reference DARTS implementation from [9] was modified, improving the runtime's performance on a wider range of applications. Some of the changes introduced in DARTS are described below.

4.1 Threaded Procedures Placement

To invoke a new TP, DARTS provides the `invoke` function. This assigns the new TP to the local TP scheduler, which controls the Micro scheduler executing the current codelet and performing the invocation call. This scheduling strategy was made to improve data locality since the codelets belonging to the new TP are likely to access the same data as the caller codelet. However, this is not always the best strategy, in particular in the case where the workload is clearly embarrassingly parallel.

To offer the option of assigning the new TP to a specific TP scheduler, either in the same or a different NUMA domain, a new function, called `place()`, was added. This function takes the same parameters as `invoke()`, plus the identifier of the target TP scheduler.

4.2 Scheduler Downtime

In the reference DARTS implementation, Micro schedulers are always either executing codelets or busy-waiting on their queue for new ready codelets. The same applies to TP schedulers, except they also have to assign codelets to their Micro schedulers. This creates an unnecessary memory trafic, resulting in a significant overhead when there is not enough work for all the schedulers. Moreover, as the core count increases on a shared-memory node, the overhead costs are amplified. This is especially true when a application launches the same DARTS runtime instance several times, executing sequential code in between (similar to having several non-nested `parallel` regions). In that scenario, all the schedulers are idly spinning around and waiting for the runtime to be launched again while the sequential code is executed. The alternative, creating a new runtime instance for every launch, is not advisable given the overhead incurred by DARTS initialization process.

To reduce the overhead produced by having schedulers in idle state, a downtime was added to each TP and Micro scheduler. We use an exponential backoff downtime strategy with a saturation cap. The downtime is reset once the scheduler has found a ready codelet to fire, or, in the case of a TP scheduler, if it has performed a resource management task, such as deleting the context of terminated TPs (see Section 4.3 below).

Currently, the values for the exponential growth and the saturation cap must be tuned by the user for the underlying architecture and passed to the runtime during execution.

4.3 TP Deletion Deferral

After executing a codelet, TP or Micro scheduler checks for any pending codelets to be fired on the same TP. If none is found, the scheduler deallocates the TP context. Performing this process continuously adds overhead to the runtime due to the underlying calls to `delete` and corresponding memory deallocations.

To alleviate this situation, a TP context deletion queue was added to each TP and Micro scheduler. When a scheduler finds a TP context that must be deleted, it adds it to the queue instead of deleting it immediately. Then, when the scheduler has no work to perform, it checks if the queue has pending TP contexts to deallocate (and proceeds if it is the case); otherwise it goes to downtime.

4.4 Affinity Environment Variable

To offer users the flexibility to pin down schedulers to the cores of their choice, the `DARTS_AFFINITY` environment variable was added to the runtime system. This variable uses the same format as the `GOMP_CPU_AFFINITY` variable available for GCC's OpenMP library (GOMP). It also includes a `verbose` option to print the physical core's ID to which each TP and Micro scheduler has been pinned down.

5 Experimental Evaluation

To evaluate the performance of the fine-grain applications generated by our environment, we used two well-known applications of high interest for the scientific community: Graph500 and NWChem-SCF. These two benchmarks were selected because they feature two distinct type of applications: one that is memory-bound with random access of memory locations (*i.e.*, Graph500) and another that is more computationally intensive with a more static memory access pattern (*i.e.*, NWChem-SCF). For each benchmark, we used a publicly available reference OpenMP code. We generated a parallel version for each, using both an OpenMP-enabled C compiler, and our *omp2cd* compiler. We then compared both versions of the benchmark.

5.1 Experimental Testbed

5.1.1 Hardware Platform

Our experiments were performed on a 4-socket shared-memory node. Each socket holds a Intel Xeon E5-4610 processor (Sandy Bridge micro-architecture), a 6-core processor with hyperthreading activated. Cores have access to a 32 KiB L1 data cache, a 32 KiB instruction cache, and a unified L2 cache. Both L1 and L2 are private. All cores share access to a 15 MiB unified L3 cache. All 48 threads have access to 128 GiB DDR3 DRAM, distributed across four NUMA domains.

5.1.2 System Software Stack

The compute node runs Ubuntu Linux (Server) v14.04. We used GCC v4.8 to compile our C and generated C++ programs, with optimizations set to `-O3`. We compared our DARTS-generated programs with the native OpenMP code generation performed by GCC.

5.2 Graph500

The Graph500 benchmark performs a parallel Breadth-First Search (BFS) in a large undirected graph starting from a randomly selected source node [13]. 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 [13] with a minor modification in the `make_bfs_tree` function: we moved the `parallel` region to the inside of the `while` loop, as the instructions that preceded the `while` in the original code can be easily executed by a single thread without loss of correctness.

Both the GCC-based and *omp2cd* versions were executed using `numactl --interleave=all` to spread the graph evenly across the different NUMA nodes of the machine. The best performance for OpenMP was obtained with `OMP_PROC_BIND=true`. For DARTS, the `for` region in the search kernel was optimized using DARTS `Loops`, as explained in Section 3.5. Also, work stealing policies and a compact core affinity mapping, which pins down TP and Micro schedulers in contiguous logical cores, were selected. Since the performance of the application is measured close to the `parallel` region of the seach kernel, the time schedulers are idle has little influence in the measurements. Consequently, the downtime feature was not used for this benchmark.

Experimental results for Graph500 are presented in Figure 3 using all the cores available in the target platform. The scale parameter, defined as the logarithm base two of the number of vertices in the graph, was varied between 14 and 26; for other parameters, their default values were used.

The fine-grain application generated by our environment is always ahead or on par with the OpenMP reference application, except for a scale value of 16, when the speedup is 0.8×. The higher speedup, 1.7×, is obtained with a scale of 26, while with a scale of 24 the speedup is 1.3×. These speedups were achieved by tuning the parameters of the `Loop` construct used in the search kernel. We tuned these parameters according to the number of neighbor vertices to visit as inputs (*i.e.*, the size of the search frontier), the total number of vertices in the graph, and the number of TP and Micro schedulers set in the runtime, which were set to match the hardware topology. The difference in performance with a scale of 16 may indicate that these parameters can be further tuned for the target platform.

Figure 3 results are consistent with those presented by Suettlerlein *et al.* in [5], which uses a hand-written DARTS BFS application. Our fine-grain application, on the other hand, has been generated by *omp2cd*, only requiring the user to tune certain parameters according to the underlying architecture. This validates our multigrain parallel programming environment as a valuable tool to reduce the time programmers need to write a fine-grain application.

5.3 NWChem-SCF

The second benchmark is Pacific Northwest National Laboratory's Self Consistent Field method, from the NWChem package software (NWChem-SCF) [14]. NWChem is a high-performance computational chemistry software that provides researchers with high-performance scalable tools for the parallel computation of large chemistry problems. SCF is an iterative fixed-point algorithm that solves a non-linear system representing the mean field created by all the particles surrounding a single one [15]. The system is solved using a Schrödinger's equation in the form of a self-consistent eigenvalue. The main routine is `twoel`, which contains calls to other functions with several `parallel` regions to compute the forces between two electrons.

The reference OpenMP code used for this benchmark is an optimized version available in [16]. The code was compiled using `BOOKKEEPING`, `OMP_VERSION=3`, and `BLAS` options. For the BLAS functions, the OpenBLAS package was used. Both versions were run using `numactl --interleave=all`, with the DARTS application executed using an affinity defined as `DARTS_AFFINITY=0-47` and work-stealing policies for all its schedulers. This scheme pins the schedulers down according to their IDs, one by one, from 0 to 47. By contrast, the compact affinity scheme used for Graph500 pins down schedulers using the sequence 0, 24, 1, 25, 3, 26 and so on.

Since the performance for this benchmark is measured around several functions, some of them with `parallel` regions and others with sequential code, the scheduler's downtime was enabled and tuned for the target platform to avoid the overhead of having the schedulers spinning idly during the execution of the sequential code.

Figures 4 and 5 present the execution times for NWChem-SCF for weak and strong scaling, respectively. The code generated with *omp2cd* was executed with both our optimized version of DARTS and the base implementation obtained from [9]. This allowed us to further evaluate the optimizations introduced to the runtime system.

When using our optimized version of DARTS, scalability grows on par with OpenMP, obtaining speedups between 0.9$\times$ and 1.1$\times$ on average; the highest being with 160 atoms (1.1$\times$), and the lowest with 100 atoms (0.9$\times$). The base DARTS version, however, does not perform that well, ranging from 0.75$\times$ (for 160 atoms) down to 0.5$\times$ (for 50 atoms) compared to the GCC/OpenMP version.

The difference between our optimized version of DARTS and the base implementation is specially noticeable in Figure 5. As we use more threads, the number of Micro schedulers is increased to match the hardware topology of the target platform. This results in an increase of the overhead of having these schedulers busy-waiting on their queues until new work arrives. In NWChem-SCF, the time during which schedulers are spinning is high because the main kernel has several `parallel` regions with calls to sequential and `BLAS` functions in between. While these are executed, the schedulers have no work to do, so they spin until the runtime is put to work again for the next `parallel` region. The downtime introduced in our optimized version allows having the schedulers in sleep mode while the sequential region is executed, reducing their overhead.

6 Related Work

Weng's Doctoral dissertation [17] proposes the translation OpenMP programs to a dataflow execution model using the Cougar compiler and SMARTS runtime system. One of the main purposes is to increase the data locality of the application by finding the best mapping, at compile time, of nodes in the task graph to processors. *omp2cd*, on the other hand, improves data locality by automatically determining the best size and number of TPs to be used for 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 that, like *omp2cd*, parses OpenMP programs to generate code for a runtime system based on a dataflow task model with different task grain sizes. The runtime, named libKOMP [18], based on XKaapi [19], uses a work-stealing dynamic scheduling algorithm and extends the features provided by OpenMP to express dataflow parallelism. The main difference between KaCC and *omp2cd* is that our compiler performs a two-level granularity analysis in the input OpenMP application given the two-level parallelism found in DARTS. *omp2cd* does not only determine the best size of each codelet, but also the best number of codelets that should be grouped into a single TP. KaCC, on the other hand, decomposes the input application into a single type of parallel constructs needed in libKOMP (*i.e.*, tasks).

OpenStream allows programmers to express dynamic dependent tasks by annotating an OpenMP application [20]. This way, programmers can specify the application's dataflow task graph 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 programming interface [21]. StarSs is a programming model that computes, at runtime, the dependencies between tasks to execute a sequential task-based program in parallel. *omp2cd* does not require extensions in order to support DARTS, and although some optimizations performed by the compiler are complemented by the underlying runtime, DARTS, *omp2cd* computes the dependencies between tasks (*i.e.*, codelets) at compile time, reducing the runtime's overhead. The fine-grain constructs and clauses added to OpenMP on its latest versions (`tasks` to v. 3.1 and `depend` to v. 4.0, respectively) that allow programmers to construct an application's task-dependence graph were in partly inspired by the work performed with OmpSs and StarSs.

7 Conclusions

We have presented the design and experimental evaluation of *omp2cd*, a multigrain parallel compiler for the translation of programs expressing parallelism with coarse-grain constructs into fine-grain multithreaded applications. The compiler, along with an optimized version of DARTS, provides a complete multigrain parallel programming environment for the execution of OpenMP programs on top of a fine-grain event-driven program execution model, like the Codelet model.

Two scientific benchmarks were used for the experimental evaluation of the fine-grain applications generated by the compiler. After tuning these applications for the target platform, their performance were on par with or better than the original OpenMP program generated by the GCC compiler. Our results generating Graph500 codelet programs from OpenMP result in speedups going up to $1.7\times$, while for NWChem-SCF, speedups range from $0.9\times$ to $1.1\times$. This shows that a fine-grain application generated by *omp2cd* can perform as well as hand-written and optimized codelet programs.

Future work will focus on incorporating in the compiler all the syntax and semantics defined in OpenMP v.2.5 to make it fully compliant, as well as the `depend` clause from OpenMP v4.0 to define dependencies among sibling tasks. Compiler hints, in the form of OpenMP clause extensions, will be also added so users can guide the compilation process with specific granularity parameters and power and resilience constraints. Finally, further evaluation of the multigrain environment will be performed with additional scientific applications.

References

[1] US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research. (2010, Fall) The Opportunities and Challenges of Exascale Computing. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

[2] ——. (2011, October) Tools for Exascale Computing: Challenges and Strategies. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/research/cs/Exascale Workshop/Exascale_Tools_Workshop_Report.pdf

[3] ——. (2012, December) Exascale Operating Systems and Runtime Software Report. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/research/cs/Exascale Workshop/ExaOSR-Report-Final.pdf

[4] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "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*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69. [Online]. Available: http://doi.acm.org/10.1145/2000417.2000424

[5] J. Suettlerlein, S. Zuckerman, and G. R. Gao, *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. An Implementation of the Codelet Model, pp. 633–644.

[6] Open Community Runtime. [Online]. Available: https://xstackwiki.modelado.org/Open_Community_Runtime

[7] ET International Inc,. SWARM (Swift Adaptative Runtime Machine). [Online]. Available: http://www.etinternational.com/index.php/products/swarmbeta

[8] clang Team. clang: a C language family Frontend for LLVM. [Online]. Available: http://www.clang.llvm.org

[9] (2016, March) The Codelet Execution Model. [Online]. Available: http://www.capsl.udel.edu/codelets.shtml

[10] *OpenMP Application Program Interface Version 2.5*. [Online]. Available: http://www.openmp.org/mp-documents/spec25.pdf

[11] *OpenMP Application Program Interface Version 3.1*. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[12] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[13] (2016, March) Graph500 Benchmark 1 ("Search"). [Online]. Available: http://www.graph500.org/specifications

[14] (2016, March) Hartree-Fock Theory for Molecules - NWCHEM. [Online]. Available: http://www.nwchem-sw.org/index.php/Release65:Hartree-Fock_Theory_for_Molecules

[15] (2016, March) NWChem SCF Module Description. [Online]. Available: https://xstackwiki.modelado.org/images/0/08/SCF_text_document.docx

[16] (2016, March) DynAX - Extreme Scale Software Stack. [Online]. Available: https://xstackwiki.modelado.org/DynAX

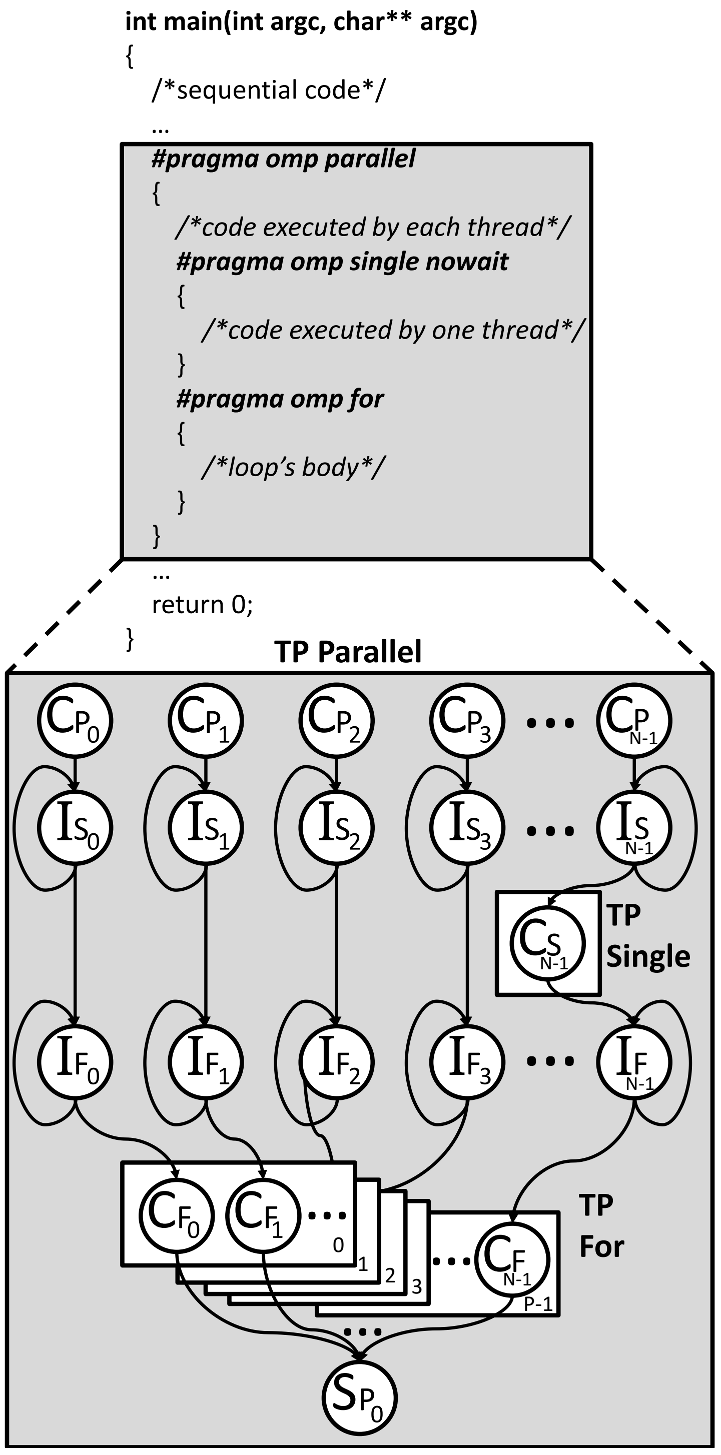
[17] T.-H. Weng, "Translation of openmp to dataflow execution model for data locality and efficient parallel execution," Ph.D. dissertation, Houston, TX, USA, 2003.

[18] F. Broquedis, T. Gautier, and V. Danjean, *OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms, pp. 102–115.

[19] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "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*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1299–1308. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.66

[20] A. Pop and A. Cohen, “Openstream: Expressiveness and data-flow compilation of openmp streaming programs,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400712

[21] Barcelona Supercomputing Center. The OmpSs Programming Model. [Online]. Available: https://pm.bsc.es/ompss



```
int main(int argc, char** argc)
{
   /*sequential code*/
   …
   #pragma omp parallel
   {
      /*code executed by each thread*/
      #pragma omp single nowait
      {
         /*code executed by one thread*/
      }
      #pragma omp for
      {
         /*loop's body*/
      }
   }
   …
   return 0;
}
```

TP Parallel
C_{P_0} C_{P_1} C_{P_2} C_{P_3} … $C_{P_{N-1}}$
I_{S_0} I_{S_1} I_{S_2} I_{S_3} … $I_{S_{N-1}}$
$C_{S_{N-1}}$
TP Single
I_{F_0} I_{F_1} I_{F_2} I_{F_3} … $I_{F_{N-1}}$
C_{F_0} C_{F_1} …
0 1 2 3
… $C_{F_{N-1}}$
P-1
TP For
…
S_{P_0}

Figure 2: Multigrain Transformation of an OpenMP Code: Rectangles represent TPs, while circles represent codelets (C for computation codelets, I for invocation codelets, and S for synchronization codelets). The first subindex on each codelet represents its `omp` region (P for `parallel`, S for `single`, F for `for`), while the second subindex represents its ID.

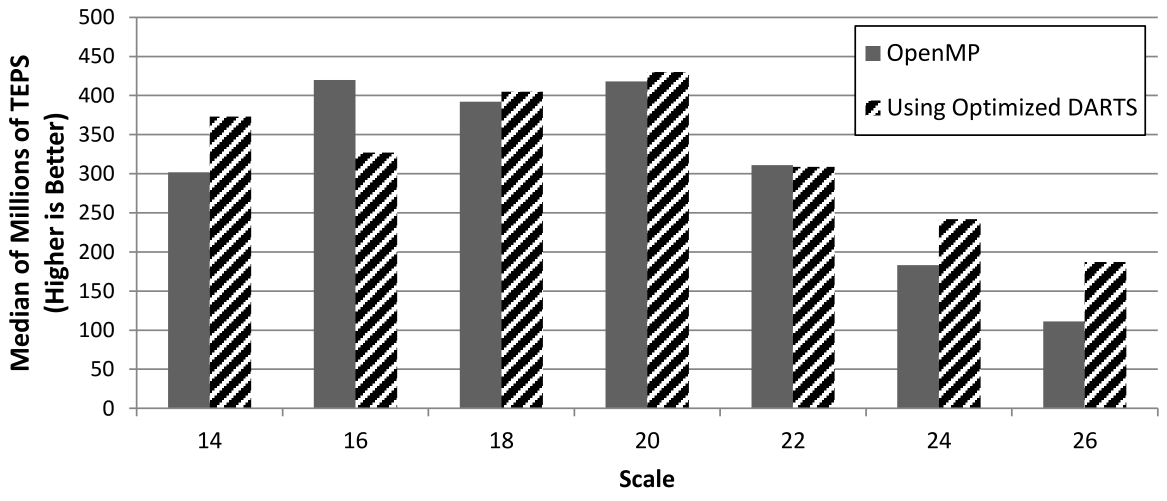
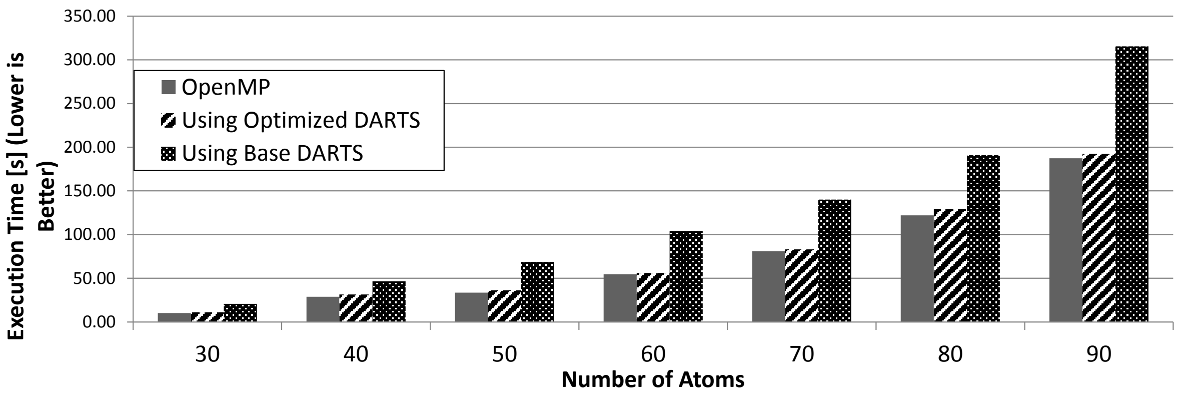
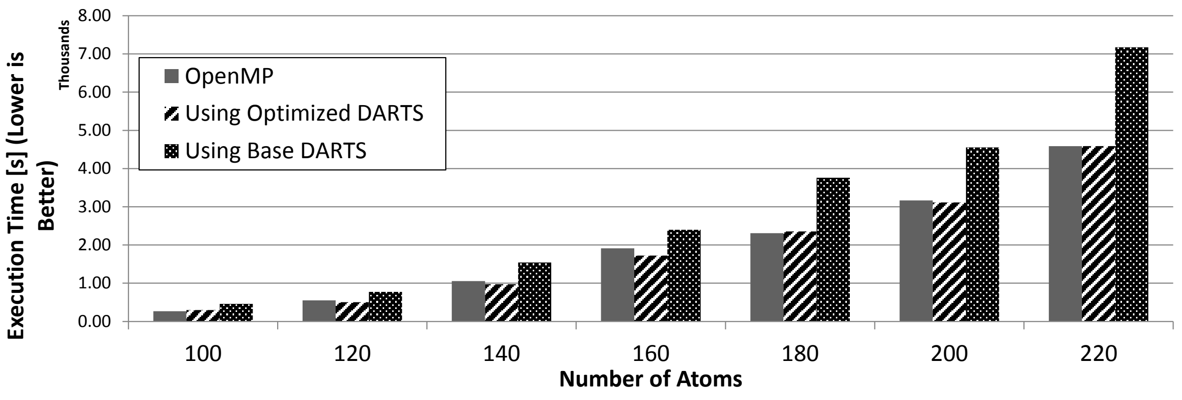


Figure 3: Graph500 Performance with varying scales and using 48 threads.



(a) Using 30 to 90 Atoms.



(b) Using 100 to 220 Atoms.

Figure 4: Weak Scalability of NWChem-SCF for 48 Threads.

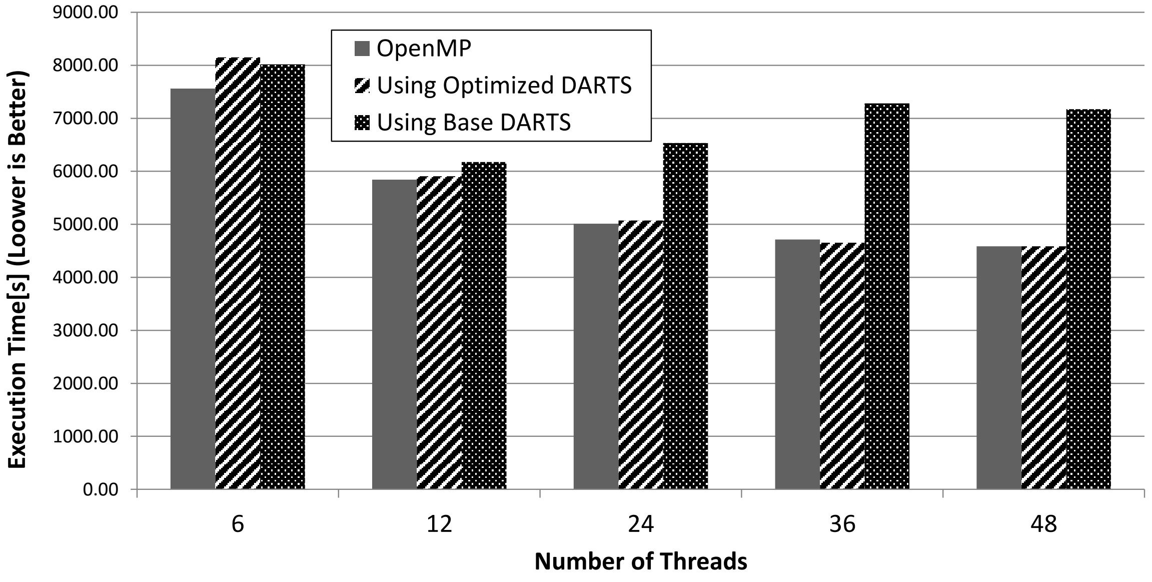


Figure 5: Strong Scalability of NWChem-SCF for 220 Atoms.