The Importance of Efficient Fine-Grain Synchronization for Many-Core Systems

Tongsheng Geng[1(✉)], Stéphane Zuckerman[2], José Monsalve[2], Alfredo Goldman[1], Sami Habib[3], Jean-Luc Gaudiot[1], and Guang R. Gao[2]

[1] PArallel Systems and Computer Architecture Lab, Department of Electrical Engineering and Computer Science, University of California, Irvine, USA
{tgeng,gaudiot}@uci.edu, gold@ime.usp.br

[2] Computer Architecture and Parallel Systems Laboratory, Department of Electrical and Computer Engineering, University of Delaware, Newark, USA
{szuckerm,josem}@udel.edu, ggao.capsl@gmail.com

[3] Computer Engineering Department, Kuwait University, Al-khalidiya, Kuwait
sami_habib@me.com

Abstract. Current shared-memory systems can feature tens of processing elements. The old assumption that coarse-grain synchronization is enough in a shared-memory system thus becomes invalid. To efficiently take advantage of such systems, we propose to use fine grain synchronization, with event-driven multithreading. To illustrate our point, we study a naïve 5-point 2D stencil kernel. We provide several synchronization variants using our fine-grain multithreading environment, and compare it to a naïve coarse-grain implementation using OpenMP. We conducted experiments on three different many-core compute nodes, with speedups ranging from 1.2× to 1.75×.

1 Introduction

In the past decade, the number of processing elements (PEs) found in general-purpose high-performance processors has increased between fourty and a hundred times, as demonstrated by, *e.g.*, Intel®'s Xeon and IBM®'s POWER8 processors. Further, so-called accelerators have reached even higher PE counts in recent years.

In the meantime, the programming models and program execution models (PXMs) used by application scientists are mostly the same: MPI is used for inter-node communication, and OpenMP is still favored for shared-memory computations. However, while the OpenMP standard has evolved to include finer-grain tasks with OpenMP 3, and even provide ways to define task-dependence graphs in OpenMP 4 [6], a large majority of application programmers still rely on a coarse-grain style to express parallelism, *i.e.*, they mostly use constructs tied to parallel `for` loops, which in turn require the use of global barriers.

While the core count remained low in compute nodes, this approach was still reasonable. However as we explained above, assuming a low core count is

© Springer International Publishing AG 2017
C. Ding et al. (Eds.): LCPC 2016, LNCS 10136, pp. 203–217, 2017.
DOI: 10.1007/978-3-319-52709-3_16

not realistic anymore. Synchronization usually leverages the use of atomic operations, which can seriously hamper performance in a multi-core, multi-socket environment. In particular, memory-bound workloads tend to tax the interconnection network linking sockets together. In general, high-performance based synchronization constructs rely on some sophisticated variation of busy-waiting (potentially mitigated with a sleep policy) which can hog the memory subsystem, as the system software designer expects contention to be low and the workload to be well-balanced—particularly in the case of embarrassingly parallel algorithms and programs. Specifically, memory-bound workloads may suffer from load imbalance due to saturated resources, *e.g.*, FPUs shared by multiple PEs, or contention on a given memory level. One such example is the use of partial differential equation iterative solvers for linear equation systems, in particular the application of Jacobi or Gauss-Seidel methods to a linear system by resorting to a stencil-based iterative solver: every element of an n-dimensional grid depends on its immediate neighbors, and potentially more remote ones. Such algorithms are used in a multitude of applications, *e.g.*, to solve Laplace equations used in heat conduction and computational fluid dynamics solvers.

In this paper, we propose to demonstrate the need for fine-grain synchronization even in the presence of rather coarse-grained workload partitioning. We compare the coarse-grain parallelization of a 5-point stencil application implemented with OpenMP to several variants using a fine-grain event-driven execution model. While there are various ways to optimize stencil codes, our intent is to demonstrate that in a dependence-heavy context, yet with a uniform amount of work per thread, *fine-grain synchronization matters*, even in “regular” general-purpose systems[1].

Our experiments show that even with a simple hierarchical scheme, the reduction in atomic operations and memory traffic in general benefits the overall execution of the program. We then further modify our variant so that parallel tasks only communicate with their neighbors. The process itself is made easier thanks to the integration in the task-definition semantics of event dependencies. Moreover, while we hand-coded our stencil computations using an implementation of the Codelet Model, the process to parallelize such a workload in a hierarchical manner is rather systematic and easy to follow.

We run our experiments on three different types of machines featuring ×86 processors, with a different number of processing elements per chip, but also a different number of sockets per node. Our results show an improvement of up to 1.75× on the speedup obtained with OpenMP.

Section 2 presents the codelet model and its runtime implementation, which we used to carry our experiments. Section 3 describes our approach to parallelize our stencil application. Section 4 describes our experimental results. Section 5 describes other work related to fine-grain multithreading and stencil computations. Finally, we conclude in Sect. 6.

[1] Note that we do *not* claim that our own environment is better than OpenMP 4.

2 The Codelet Model

The Codelet Model [21] is a fine-grain event-driven program execution model which targets current and future multi- and many-core architectures (A short introduction is available at http://www.capsl.udel.edu/codelets.shtml). In essence, it is inspired by dataflow models of computation [8].

2.1 General Principles

Codelets: Definition and Firing Rules. The quantum of execution is the *codelet*, a fine-grain task that executes a sequence of machine instructions until completion, and runs on a von Neumann type of computation core. A codelet *fires* when all its dependencies (data and resource requirements) are met. A codelet cannot be preempted while it is firing, *i.e.*, while it is executing on a computation core.

Codelet Graphs and Threaded Procedures. Each time a codelet produces data items or releases a shared resource, it signals the other codelets that depend on such data item(s) and/or resource(s). Such a group of codelets and their dependencies can be modeled as a directed graph called a *codelet graph* (CDG). In general, a given CDG statically specifies the dependencies between the codelets it contains.

A *Threaded Procedure* (TP) is a container that comprises a CDG and data to be accessed by the codelets it contains. A TP is essentially an asynchronous function: once it has invoked a TP, its caller resumes its execution. The TP itself can run anywhere on the machine once it has been scheduled for execution.

2.2 The Codelet Abstract Machine

The codelet model relies on a Codelet Abstract Machine (CAM), which models a general purpose many-core architecture with two types of cores: synchronization units (SUs) and computation units (CUs). A CAM is composed of clusters of cores: each cluster contains at least one SU, one or more CUs, and some local memory. Clusters are grouped together to form a chip, which itself has access to some memory modules. Multiple chips can be grouped into a node, and multiple nodes form a full machine. At each level of the hierarchy, an interconnection network is assumed in order to allow for memory transfers.

A CAM is meant to be mapped on real hardware: the number of clusters, and computation units per cluster will be directly influenced by the actual hardware architecture on which a codelet program should be running. Further, different configurations may be used on the same target hardware, depending on the nature of the application.

2.3 A Codelet Runtime System

Our work relies on `DARTS`, a faithful implementation of the codelet model [19]. It targets shared-memory nodes (there is no distributed memory implementation at the time of this writing). `DARTS` executes on regular multi-core chips and assigns a role to each core: a core is either a synchronization unit or a computation unit.

3 Applying Fine-Grain Parallelism to Embarrasingly Parallel Problems

This section describes how we started from an OpenMP coarse-grain implementation of a simple, naive 5-point stencil computation and reproduced its overall structure using a codelet runtime system, to gradually refine the stencil code parallelization and leverage finer-grain synchronization.

3.1 Basic Implementation of a Parallel Coarse-Grain 5-Point Stencil

The code presented in Listing 1.1 is a naïve OpenMP version of a coarse-grain multithreaded 5-point stencil computation. To simplify the problem, we do not consider the convergence test and only rely on a given number of time steps. This version of the stencil code privatizes everything, so that each thread can perform all computations (including pointer swapping and moving forward to the next time step). The computation itself is located in a parallel `for` loop (see line 15). We removed the implicit barrier at the end of the loop so that threads that finish processing their own iteration chunk may proceed to swap their source and destination pointers for the next time step. The only required synchronization is the global barrier (line 17) before looping to the next iteration in the `while` loop, to ensure that all threads have properly swapped their array pointers before resuming the computation.

```
void  stencil_5pt(double* restrict dst,    double* restrict src,                 1
                  const size_t     n_rows, const size_t     n_cols,              2
                  size_t           n_steps)                                      3
{                                                                                4
  typedef double (*Array2D)[n_cols];                                             5
# pragma omp parallel default(none) shared(src, dst) \                           6
             firstprivate(n_rows, n_cols, n_tsteps)                              7
  {                                                                              8
    Array2D  D = (Array2D) dst, S = (Array2D) src;                               9
    size_t n_ts  = n_tsteps;                                                     10
    while (n_ts-- > 0) {                                                         11
#     pragma omp for nowait                                                      12
      for (size_t i=1; i<n_rows-1; ++i)                                          13
        for (size_t j=1; j<n_cols-1; ++j)                                        14
          D[i][j] = 0.25 * (S[i-1][j]+S[i+1][j] + S[i][j-1]+S[i][j+1]);         15
      SWAP_PTR(&D,&S);                                                           16
#     pragma omp barrier                                                         17
    }                                                                            18
  }                                                                              19
}                                                                                20
```

Listing 1.1. Naïve 5-Point Stencil kernel—OpenMP version. Everything has been privatized, but threads can only proceed to the next time step if they all have swapped their array pointers.

We first adapted the code of Listing 1.1 to our `DARTS` framework. The definition of codelets and threaded procedures is shown in Listing 1.2. The codelets are defined with default dependence counts (0 for `Compute`, and 2 for `Barrier`), but they can be overriden when they are effectively instantiated. The `Stencil` TP is essentially a C++ `struct` which allocates the right amount of codelets for a given cluster of cores, and holds the data which the codelets can access.

Table 1. Codelet Model macros and their meaning.

Keyword	Description
`DEF_TP`	Defines a new threaded procedure
`DEF_CODELET`	Defines a new codelet
`DEF_CODELET_ITER`	Defines a new codelet with a specific ID
`SYNC`	Signals a codelet within the same TP frame
`SIGNAL`	Signals a codelet in another TP frame
`SIGNAL_CODELET`	Signals a codelet from a TP setup phase
`LOAD_FRAME`	Loads the threaded procedure frame
`FIRE(CodeletName)`	Code to run when `CodeletName` is fired
`INVOKE(TPName,...)`	Invokes a new TP from a codelet

The listing of the first variant we implemented, which we call `Naive` in our experiments (see Sect. 4), is not shown here due to lack of space. The `Compute` codelet proceeds to execute the stencil operation for one time step over a chunk of the data. When it is done firing, it signals the `Barrier` codelet, which collects all the signals of all firing `Computes`. `Barrier` then proceeds to invoke a new `Stencil` TP where the source and destination arrays are swapped in the parameters list, and the time step is decreased. This variant performs poorly compared to OpenMP, as we require `DARTS` to allocate a new codelet graph for each new time step. The second variant still implements a coarse-grain synchronization scheme, but this time, it has `Compute` codelets reset their dependence count when they are fired. `Barrier` signals the end of the computation if there are no more time steps, or it resets itself, and then signals `Compute` codelets. The code is provided in Listing 1.3[2].

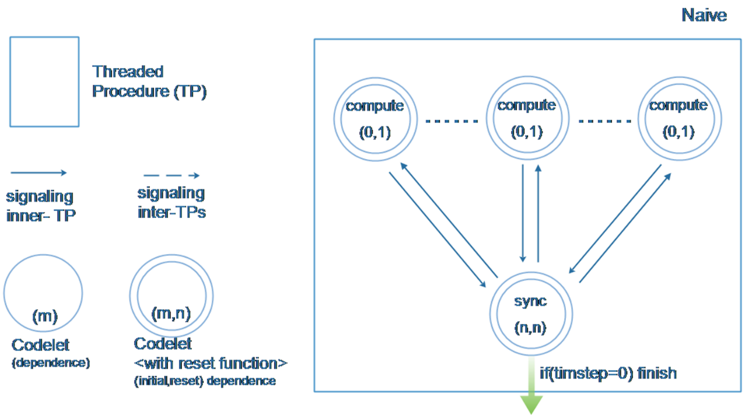

Fig. 1. A coarse-grain version of a naïve stencil computation. Each codelet resets itself if there are remaining iteration steps.

[2] Obviously, as we are writing directly using a runtime system API, the code has to be more verbose than its OpenMP counterpart.

The various keywords emphasized in bold red are macros defined to simplify the writing of DARTS programs. A short description of the various keywords is provided in Table 1. A graphical illustration of the codelet program (Naïve Stencil-DARTS with reset function) is shown in Fig. 1.

```
DEF_CODELET_ITER ( Compute , 0, NO_META_DATA );
DEF_CODELET      ( Barrier , 2, NO_META_DATA );
DEF_TP(Stencil) {
// Data
  double *dst, *src;
  size_t  n_rows, n_cols, n_tsteps;
// Code
  Compute* compute;
  Barrier  barrier;

  Stencil(double* restrict p_dst,   double* restrict p_src,
          size_t           p_nRows, size_t           p_nCols,
          size_t           p_nTSteps)
  : dst(p_dst), src(p_src)
  , n_rows(p_nRows), n_cols(p_nCols), n_tsteps(p_nTSteps)
  , compute(new Compute[g_nCU])
  , barrier(g_nCU,g_nCU,this,NO_META_DATA)
  {
    for (size_t cid = 0; i < g_nCU; ++cid) {
      compute[cid] = Compute{1,1,this,NO_META_DATA,cid};
      SIGNAL_CODELET(compute[cid]);
    }
  }
};
```

Listing 1.2. Coarse-Grain 5-Point Stencil kernel—DARTS version. `Stencil` TP definition and its associated codelets.

```
FIRE(Compute) {
  LOAD_FRAME(Stencil);
  typedef double (*Array2D)[n_cols];
  Array2D D = (Array2D) FRAME(dst), S = (Array2D) FRAME(src);
  const size_t n_rows  = FRAME(n_rows), n_cols  = FRAME(n_cols),
               n_steps = FRAME(n_steps);

  size_t cid = getID(), // current codelet's ID
         lo  = lower_bound(n_cols,cid),
         hi  = upper_bound(n_cols,cid);

  RESET(compute[cid]);
  for (size_t i = lo; i < hi-1; ++i)
    for (size_t j = 1; j < n_cols-1; ++j)
      D[i][j] = 0.25 * (S[i-1][j]+S[i+1][j] + S[i][j-1]+S[i][j+1]);
  SYNC(barrier);
  EXIT_TP();
}

FIRE(Barrier) {
  LOAD_FRAME(Stencil);
  if ( FRAME(n_tstep) == 0 ) SIGNAL(done), EXIT_TP();

  RESET(barrier);
  for (size_t i = 0; i < g_nCU; ++i) SYNC(compute[i]);
  EXIT_TP();
}
```

Listing 1.3. Coarse-Grain 5-Point Stencil kernel—DARTS version. Codelets reset themselves until the last iteration step is reached.

3.2 Description of Parallel Stencil Computation Variants

Distributing the Computation Over Multiple Clusters in the Codelet Abstract Machine. The code presented in Listing 1.3 is sufficient in case we map a codelet abstract machine (CAM) which features only a single Synchronization Unit (SU, see Sect. 2). However, this configuration centralizes all codelet graph creations onto a single processing element. Further, it creates a single unique synchronization object which will be accessed by all codelets to signal the end of their computation. This will force the whole compute node to serialize memory accesses when performing the synchronization step. As a result, we implemented a new variant inspired by the very first naïve one, which partitions the codelet graph into sub-graphs, and each contained within its own threaded procedure featuring a local `Barrier` codelet, and each confined to a given cluster of cores to maintain locality. Note that, following the original code, new TPs *are* invoked for each new time step in the computation. However, to avoid paying the cost of dynamically allocating the various codelets involved per cluster, the same array of codelets is passed from invocation to invocation: the codelets are destroyed only once the last iteration step has been reached. Figure 2 provides a high-level view of the resulting codelet graph.

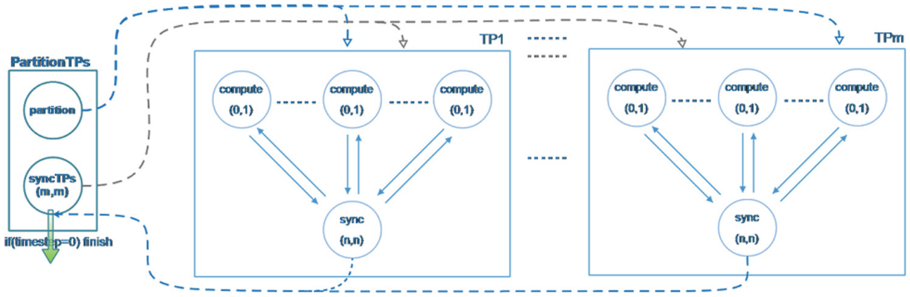

Fig. 2. A medium-grain version of a naive 5-point stencil computation. The computation is decomposed into several sub-codelet graphs, allowing a machine to hold multiple synchronization units for a better workload balance.

Toward a Finer-Grain Approach. Our goal is to allow portions of work to proceed with the next iteration step, as long as the shared rows they require to update their portion of the matrix are up-to-date. We are still decomposing the work along the rows of the matrices, but this time, each codelet simply signals its neighbors when it is done updating the rows they depend on to move to the next iteration step. Hence, some codelets may proceed to update the system at step S_{t+1} while others are still finishing step S_t. Figure 3 provides a diagram of the resulting codelet graph. In this case, we create a single TP holding the whole codelet graph, where all dependencies are statically determined. The stress on the memory subsystem is not expected to be excessive, since signals are now only sent between "neighboring" cores, thus confining atomic operations to PEs that are physically close.

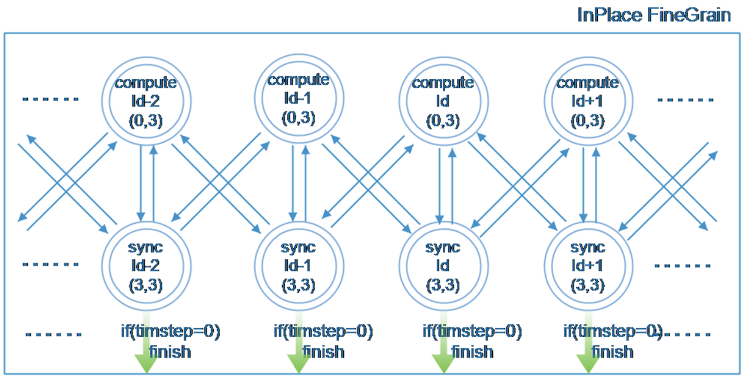

Fig. 3. A fine-grain version of a naive stencil computation. A single TP is generated, which holds the full codelet graph. Codelets only signal the neighbors which read and write shared rows.

Reducing the Stencil Computation's Footprint. The fine-grain approach we followed in the previous section also makes it easier to reduce the memory footprint of the computation. Rather than systematically using two matrices to iteratively compute new values at each time step (subsequently requiring to exchange array pointers), it is possible to allocate a small buffer per codelet in each invoked TP. Each buffer must be large enough to hold a set of at least three full rows in the matrix. The original naive loop thus becomes more complex, as each codelet must now first write the new values of the system to its local buffer first, then must write the newly updated row(s) back to the original matrix. However, this scheme lends itself well to fine-grain synchronization. Indeed, as Fig. 3 only features TPs, codelets, and their dependencies, but not the actual code or data that are held in the TP frames, then it is also an adequate representation of an "in-place" version of a fine-grain version of an n-point stencil computation. However, this version suffers from the same limitation as the previous fine-grain variant: it requires to invoke a single threaded procedure, thus forcing the codelet abstract machine to be mapped with a single SU for the whole machine, and, in turn, to accept that all TP creations will involve a potentially heavy serial step.

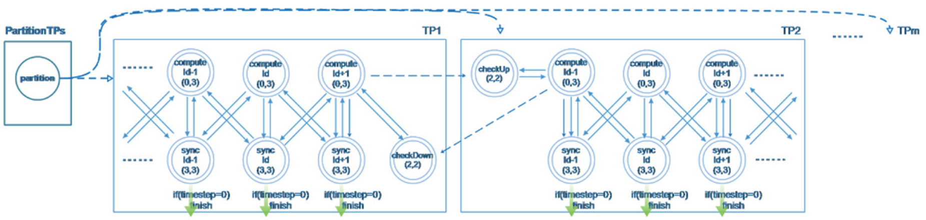

Fig. 4. A fine-grain in-place version of a naïve stencil computation. Multiple TPs can be generated, which hold a portion of the overall codelet graph. Codelets only signal the neighbors which read and write shared rows. A single matrix is required.

Hence, a final refinement is to allow for the distribution of the fine-grain "in-place" variant over multiple TPs. While the previous variants, including the initial fine-grain one, were relatively easy to implement, this specific implementation requires some careful coding when setting up the overall codelet graph, as codelets will reset themselves and signal each other not only within the same TP frame, but also *across* frames. However, the basic structure remains the same, and it clearly can be automated by a compiler. The resulting codelet graph is shown in Fig. 4. In this last variant, each codelet graph features three types of codelets: `Compute` performs the actual computation, as before. The `CheckDown` and `CheckUp` codelets are signaled when rows shared by "upper" and/or "lower" neighbors are ready to be updated. In turn, they also signal other compute codelets to let them know that the rows they are sharing with their neighbors are cleared for reading.

4 Experimental Results

4.1 Experimental Setup

The hardware platforms characteristics are described in Table 2.

Table 2. Compute nodes characteristics. "PE" = "Processing element." L2 and L3 caches are all unified. Hyperthreaded cores feature two threads per core. Platform A features 64 GiB of DRAM; architectures B and C feature 128 GiB.

Platform	Processor type	# Sockets	# PEs per Socket	Total PEs	L1D (KiB)	L2 (KiB)	L3 (MiB)	Comments
A	Intel Sandy Bridge	2	16	32	32	256	20	Private L2; hyper-threading
B	Intel Sandy Bridge	4	12	48	32	256	15	Private L2; hyper-threading
C	AMD bulldozer interlagos	4	12	48	16	2048	12	L2 & FPU are shared by 2 cores

Table 3 provides the information related to the system software running on each compute node where we ran our experiments. Each platform offers a relatively varied system software layer, with compilers and OS kernels being slightly (or even widely) different from node to node. All experiments are run by pinning threads to a given processing element (hardware thread or core), by setting the `OMP_PROC_BIND` environment variable to `true` (for OpenMP). `DARTS` automatically pins its work queues to the underlying processing elements.

Table 3. System software stack used for the experiments.

Platform	Linux distribution	Kernel version	GCC version
A	CentOS 7.1	3.10.0	4.8.3
B	Ubuntu 14.04.3 LTS	3.13.0	4.8.4
C	Scientific Linux 6.1	2.6.32	4.9.3

4.2 Experimental Protocol

We ran seven different variants of our stencil code: `Seq` is our baseline and is a benchmark that runs sequentially; `OMP` runs the same code as `Seq` with added OpenMP directives; `Naive` is a single threaded procedure implementation of the stencil computation (see Sect. 3.1), `NaiveTPsPtr` implements the same logic as `Naive`, but distributes the work across several TPs; `FineGrain` implements the fine-grain synchronization scheme described in Sect. 3.2; `InPlace` implements our in-place strategy to run the stencil computation, using a single TP; and `InPlaceTPs` implements the same in-place variant, but distributes the computation across multiple TPs which then must issue inter-TP signals to satisfy dependencies.

We ran our experiments using the following protocol: (1) All stencil computations run for 30 time steps, (2) Each variant instance is run 20 times to increase the stability of the run, then the accumulated times are averaged after removing the 2 most extreme values (min and max), and (3) Each binary containing a variant is run 10 times from the command line, and we average the accumulated times once again (this is due to system-induced noise in sequential, codelet, or OpenMP variants—in particular for small input sizes).

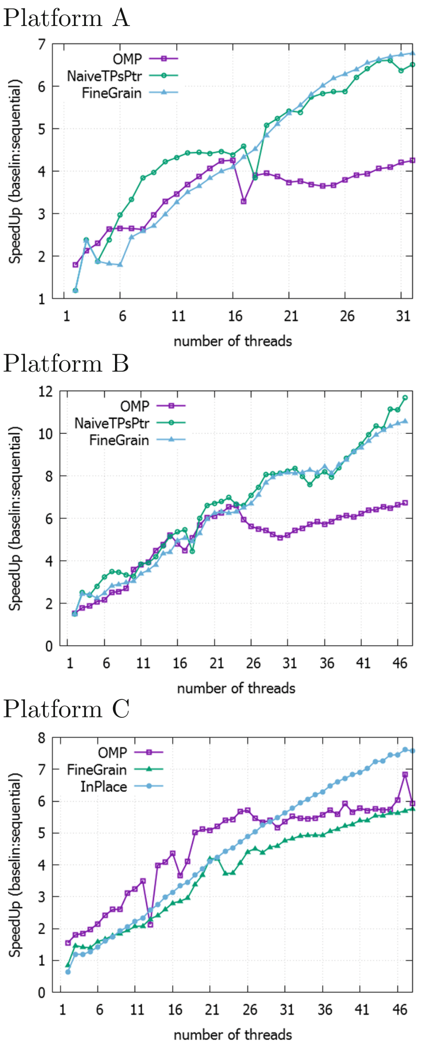

Fig. 5. 5-point 2D stencil. Strong scaling for a 3000×3000 input matrix. The baseline is the pure sequential code. We only show the two best performing `DARTS` variants.

4.3 Results

The results for strong scaling are shown in Fig. 5. The default CAM is used in the case of `DARTS`, which maps compute units to PEs that are physically close to each other. As a result, we do not use the entirety of the available aggregated cache capacity. In the OpenMP case, we used `OMP_PROC_BIND`, to make sure that threads are pinned to a given PE. However they are assigned in a more random fashion (left to the discretion of the OpenMP runtime and the OS), thus making better use of the overall caches. Still, when resources start to be saturated, *i.e.*, when more than half of the processing elements are used, and start to compete for FPUs, caches, *etc.*, the `DARTS` variants outperform the OpenMP version. As the PE count increases, so does the performance gap.

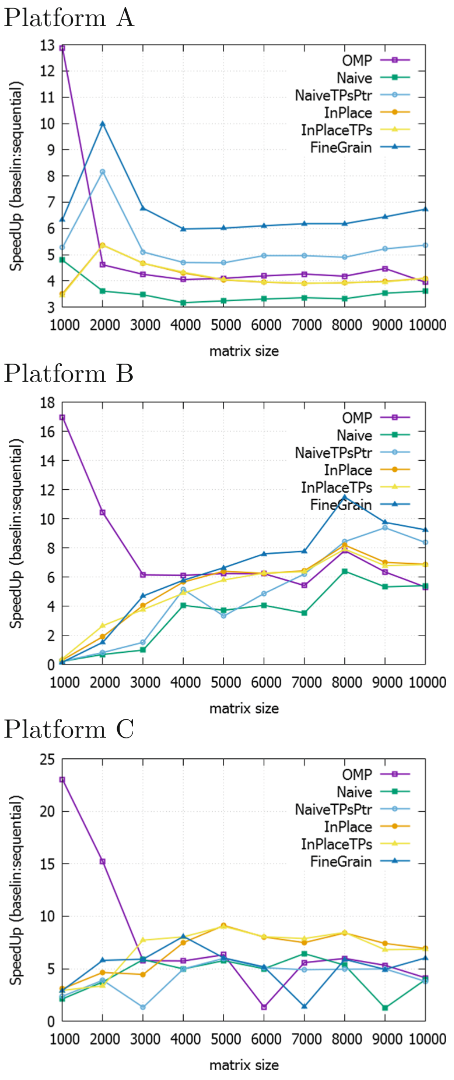

Fig. 6. 5-point stencil computation. Weak scaling. The baseline is the pure sequential code.

In the weak scaling case, the results for all variants are shown in Fig. 6. As with the strong scaling case, `FineGrain` and `NaiveTPsPtr` achieve the best performance on Intel-based architectures (A and B), with speedups reaching up to 1.75× compared to OpenMP. The OpenMP variant has a clear advantage over `DARTS` when the workload fits in the caches (*i.e.*, when the matrix size is 1000, or possibly 2000, as it still partially fits in the caches). In the OpenMP case, loops are statically scheduled, thus ensuring that the same PE processes the same chunk of data, thus minimizing cache misses. In contrast, codelets in `DARTS` can be run by any PE belonging to the same cluster of cores. Hence a given data chunk may be processed by different PEs over two successive iteration steps, resulting in additional cache trashing.

Once the data grows beyond the capacity of L3 caches, `DARTS` gets the upper hand: the finer-grain variants either issue "local" atomic operations between neighbors (as with the `FineGrain` variant), or at least provide a hierarchical way to maintain some locality within their cluster of cores, thus reducing the overall memory traffic. In particular in the Intel compute nodes (Platforms A and B), the inclusive nature of the caches allows the hardware to recognize when a given memory location is owned by the "local" L3, and thus avoids a costly request for ownership across sockets.

Unfortunately, Platform C features exclusive caches, thus forcing the hardware to issue a broadcast to flush write buffers across the whole node, as it does not know which other caches own a copy of the data [17].

4.4 Discussion

Coarse-grain synchronizations (*e.g.*, barriers) tend to be implemented with a single memory location. This has several negative consequences: (1) all processing elements issue an atomic operation to the same location, forcing the other PEs to flush their write buffers, sometimes more than once; (2) there is a "natural" contention due to the target single location. By contrast, finer-grain synchronization makes use of more locations with better locality effects. Write buffer flushes still occur, but tend to be limited to writing back in L3 (at least in the Intel case). In addition, codelets can better exploit the "slack" that exists when a core is done running a thread, due to their event-driven nature.

Finer-grain synchronization clearly *does* provide better results on general-purpose many-core systems, as shown in Figs. 5 and 6. However, which variant works best varies significantly depending on which platform we run our tests. On Intel-based compute nodes, our most refined variants did not perform very well in the end: the `InPlace` and `InPlaceTPs` variants underperformed compared to their most simple counterparts, and even compared to the coarse-grain OpenMP version. We attribute this to too naive an implementation: while the `InPlace` variant does require less memory than the original code, its implementation is too simplistic: it makes use of dynamic allocation each time a computation codelet is being fired, which in turn invokes the OS to perform the allocation itself. As most codelets are fired within a very small time range, some serialization while trying to access the OS's memory allocator results in wasted time. As Intel-based nodes feature inclusive caches, the data can only be as big as the L3s of the system.

By contrast, as Platform C is AMD-based, caches are exclusive: the aggregated size of the L2 caches equals the aggregated size of the L3s, effectively doubling the overall size of the data that can be held in the caches. It also helps with the `InPlace` and `InPlaceTPs` variants, as the local buffer allocated for the fine-grain update of the matrix is held in a separate cache than the original matrix. This is compared to the naïve, 2-array version which requires to constantly read and write from and to memory through the L1 and L2 caches. The AMD system also relies on write-through L1D caches (compared to Intel's write-back L1Ds), which allows for a better utilization of the L1D (there is roughly four times more reads than writes in the stencil computation).

Moreover, as we intended to show the benefits of "pure" fine-grain synchronization, without resorting to classical loop transformations, such as tiling or loop skewing, even the allocation of just three complete rows is enough to quickly fill L1D caches. For example, our smallest input size for a matrix, 1000×1000, requires three rows of a thousand elements to implement the current in-place variants. However, this represents already 2/3 of the L1D cache of the Intel-based compute nodes, and overflows into the L2 cache in the case of the AMD compute node. Hence, to obtain an efficient in-place variant, additional blocking and tiling techniques are required. We intend to explore this research venue, but to be fair to coarse-grain models, we must do the same for the naïve OpenMP code.

5 Related Work

Fine-Grain Multithreading Program Execution Models. In recent years, several attempts at providing more dynamic ways to create parallel work have been proposed. Many such attempts are inspired by dataflow models of computation. Among them, we can mention Concurrent Collection [11], an implementation of dynamic macro-dataflow. It has shown encouraging results, including on stencil-like computations [14]. XKaapi [9], OCR [15], and SWARM [12] all propose a dataflow (or even codelet) inspired way to deal with multithreading. However, they do not provide an explicit way to group dataflow tasks to ensure they execute on a specific portion of the hardware (for example, to maintain spacial and temporal locality), contrary to `DARTS` (which uses threaded procedures to enforce codelet grouping). Other frameworks provide fine-grain multithreading without being directly tied to dataflow. Chief among them are Cilk [5] and Habanero [3].

Finally, the latest version of the OpenMP standard proposes a way to describe task dependencies in a program [6], by describing dataflow-like dependencies in the code. The resulting task dependence graph is obtained in a fully dynamic manner. By contrast, `DARTS`'s codelet graphs tend to dynamically allocate chunks of codelets which feature statically-defined dependencies.

Frameworks and Transformations for Stencil Computations. While this paper's intent is to advocate for finer-grain synchronization for large-scale general-purpose compute nodes, and uses stencil kernels only as an example, we provide a short description of related techniques and frameworks to optimize stencil computations.

Classical loop optimization techniques provide very efficient ways to improve sequential stencil computation. Loop tiling, locality optimization and parallelization are the main methodology to improve stencil computation performance. Loop tiling [1] manipulates hyperplanes from the iteration space to determine the tile shapes for a given computation, as well as the scheduling order. Further transformations include diamond tiling [2,4]. More recently, the manipulation of the iteration space has led to better work scheduling for many-core devices. For example, Shrestha *et al.* propose to perform transformations on the iteration space using jagged-tiling to allow for a better concurrent start for processing tiles in parallel [18].

Pochoir is a domain-specific language relying on Cilk that allows the user to specify a given type of stencil computation to be generated automatically for parallel execution [20]. Kamil *et al.* [7,10] propose a code generation and auto-tuning framework for stencil computations targeted at multi- and many-core processors. Muranushi and Makino introduced the PiTCH tiling method [16], which leverages a temporal blocking methodology which can achieve a target's optimal memory bandwidth ratio well-suited for multidimensional stencil computations. Lesniak introduced a block-based wave-front synchronization technique for parallel stencil calculation [13].

6 Conclusion and Future Work

We have presented a study of a dependence-heavy application to advocate for finer-grain and hierarchical synchronization in current high-performance general purpose many-core compute nodes. Leveraging a runtime system implementation of a fine-grain event-driven execution model, we have devised several variants to study the best way to leverage fine-grain synchronization, and demonstrated that by using finer-grained synchronization, even embarrassingly parallel workloads can see their performance improve by up to 1.75× using regular work distribution among cores.

Our future work includes rewriting the original naïve OpenMP code using OpenMP 4.5's task dependence constructs, and compare the resulting performance with our own environment's. While the fine-grain variants we have presented in this paper were hand-written, most of them can be implemented in a compiler, using a syntax close or identical to OpenMP 4's. We are in the process of developing a compiler that translates OpenMP code to fine-grain event-driven tasks, and generates automatically a multi-level synchronization scheme—we believe OpenMP's programming model is enough to express parallelism, but that the Codelet Model provides a better program execution model.

Acknowledgments. This research is based upon work supported by the National Science Foundation, under awards XPS-1439165 and XPS-1439097.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. SIGPLAN Not. **26**(7), 39–50 (1991)
2. Bandishti, V., Pananilath, I., Bondhugula, U.: Tiling stencil computations to maximize parallelism. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012. IEEE Computer Society Press, Salt Lake City (2012)
3. Barik, R., et al.: The Habanero multicore software research project. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009. ACM, Orlando (2009)
4. Bertolacci, I.J., et al.: Parameterized diamond tiling for stencil computations with chapel parallel iterators. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015. ACM, Newport Beach (2015)
5. Blumofe, R.D., et al.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)

6. OpenMP Architecture Review Board. OpenMP Application Program Interface version 4.0 (2013)
7. Christen, M., Schenk, O., Burkhart, H.: PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS) (2011)
8. Dennis, J.B.: First version of a data flow procedure language. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 362–376. Springer, Heidelberg (1974). doi:10.1007/3-540-06859-7_145
9. Gautier, T., et al.: XKaapi: a runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS) (2013)
10. Kamil, S., et al.: An auto-tuning framework for parallel multicore stencil computations. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS) (2010)
11. Knobe, K.: Ease of use with concurrent collections (CnC). In: Hot Topics in Parallelism (2009)
12. Lauderdale, C., Khan, R.: 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 Exafop Era, EXADAPT 2012. ACM, London (2012)
13. Lesniak, M.: PASTHA: parallelizing stencil calculations in Haskell. In: Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP 2010. ACM, Madrid (2010)
14. Liu, C., Kulkarni, M.: Optimizing the LULESH stencil code using concurrent collections. In: Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frame-Works for High Performance Computing, WOLFHPC 2015. ACM, Austin (2015)
15. Mattson, T., et al.: OCR: the open community runtime interface. Technical report, June 2015. https://xstack.exascaletech.com/git/public
16. Muranushi, T., Makino, J.: Optimal temporal blocking for stencil computation. Procedia Comput. Sci. **51**, 1303–1312 (2015). International Conference on Computational Science, ICCS 2015 Computational Science at the Gates of Nature
17. Schweizer, H., Besta, M., Hoefler, T.: Evaluating the cost of atomic operations on modern architectures. Technical report ETH Zurich, Department of Computer Science (2015)
18. Shrestha, S., Manzano, J., Marquez, A., Feo, J., Gao, G.R.: Jagged tiling for intra-tile parallelism and fine-grain multithreading. In: Brodman, J., Tu, P. (eds.) LCPC 2014. LNCS, vol. 8967, pp. 161–175. Springer, Heidelberg (2015). doi:10.1007/978-3-319-17473-0_11
19. Suettlerlein, J., Zuckerman, S., Gao, G.R.: An implementation of the codelet model. In: Wolf, F., Mohr, B., Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 633–644. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40047-6_63
20. Tang, Y., et al.: The pochoir stencil compiler. In: Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011. ACM, San Jose (2011)
21. Zuckerman, S., et al.: 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 2011. ACM, San Jose (2011)