Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Hybrid Use of OmpSs for a Shock Hydrodynamics Proxy Application

Jan Christian Meyer[a]*

[a]*High Performance Computing Section, IT Dept., Norwegian University of Science and Technology*

Abstract

The LULESH proxy application models the behavior of the ALE3D multi-physics code with an explicit shock hydrodynamics problem, and is made in order to evaluate interactions between programming models and architectures, using a representative code significantly less complex than the application it models. As identified in the PRACE deliverable D7.2.1 [1], the OmpSs programming model specifically targets programming at the exascale, and this whitepaper investigates the effectiveness of its support for development on hybrid architectures.

Introduction

With the largest contemporary parallel systems featuring specialized accelerator units to address the challenges of approaching exascale, OmpSs is a programming model that aims to support programming such units, and was identified as a candidate exascale technology in the PRACE deliverable D7.2.1 [1]. It represents an ongoing effort of the programming models group at the Barcelona Supercomputing Center [2] to integrate features of their previous StarSs programming model into a single model. OmpSs extends OpenMP [3] with directives to support asynchronous parallelism and heterogeneous platforms. One of the main additions of OmpSs is the particular treatment of tasks, introduced in OpenMP 3.0 As it is an extension of OpenMP, programs parallelized using OpenMP can be run without significant modifications.

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code (LULESH) [4] is a proxy application developed at Lawrence Livermore National Labs, as a tool for investigating co-design efforts investigating programming models and architectures to target exascale performance. It is a highly simplified application which has been hard-coded to solve one particular blast problem which has an analytic solution, but includes the relevant numerical algorithms, data movement characteristics and programming style of a full application. Its utility lies in evaluating the fitness of programming models and architectures without requiring extensive porting efforts, as the reference implementation is an easily managed, and relatively small source code.

Background and Motivation

Because the OmpSs programming model is an extension of OpenMP, and bears a strong resemblance to it, it is most interesting to examine the influence of the points where they differ. Syntactically, the models are practically identical, so the effort to adapt an application from one to the other is mainly concerned with program semantics and their interactions with the run time system. Since LULESH was already ported to OpenMP, that port provides a natural starting point for adaptation to OmpSs. Two main points are considered, specifically, the use

* Corresponding author. *E-mail address*: Jan.Christian.Meyer@ntnu.no

of work-sharing directives, and the potential for using task definition as a simplified means of transferring computation to graphics accelerators.

Work-sharing Directives in a Task-based Environment

The Nanos run-time system which handles the execution of OmpSs programs is inherently task-based, and supports parallelism through a programming interface which is used by the Mercurium compilers to translate pragma directives. The run-time layer constructs a directed, acyclic dependency graph of spawned tasks, and schedules their execution subject to the constraints it encodes. User programs can specify these dependencies explicitly through using input/output clauses which extend the *task* directive of OpenMP. The translation of work-sharing directives such as independent loop iterations or parallel sections is implicitly handled by *slicers* in the run time system, which spawn tasks that partition the iterations or regions depending on their specification. Although work-sharing directives were chosen for OpenMP, the OmpSs design decision to treat these as a means of generating tasks requires a re-examination of LULESH performance trade-offs, to ensure that the style of parallelism most appropriate to OmpSs is chosen for an SMP implementation.

Task Directives to Enable Accelerator Use

Aside from the use of run-time tasks as a more fine-grained mechanism to control program synchronization requirements, the *task* directive in OmpSs also features extensions which permit tasks to be implemented by computational kernels targeting accelerator devices. This effort to produce a unified calling interface useful for multiple architectures is worthwhile to investigate not only because of the potential for improved scalability from enabling hybrid programs, but also because of the resource requirements of maintaining multiple stand-alone programs to cover various architectures.

The following sections describe experiments conducted to evaluate how OmpSs addresses these points.

SMP Case Study

One of the main differences between OpenMP and OmpSs is that while the former requires *parallel* directives to spawn parallel thread teams throughout run time, OmpSs runs with constant team throughout program execution, and schedules tasks to its members as they are created. For this reason, the *parallel* directives of OpenMP are effectively ignored by the OmpSs toolchain. Accordingly, starting from the LULESH OpenMP code, the first step was to remove three parallel regions containing sequences of non-blocking loops.

Two specific parallelized loops required removal of their initial parallelization altogether, as compilation halted with an error message related to the naming scheme of the loop bodies' internal representation as tasks: these loops were found in the *CalcHydroConstraintForElems* and *CalcCourantConstraintForElems* subroutines. Because the small part these loops represent in the total computation time, they were left in serialized state to guarantee correctness, and workarounds for the issue were not pursued further. This produced a functioning initial implementation for SMP/multi-core execution, which was verified to produce the correct values of final origin energy for the common problem sizes 5^3, 10^3, 45^3, 50^3, 70^3 and 90^3, listed by Karlin *et al.* [4].

Task Selection

The performance consideration that spawning tasks implies a creation overhead which must be amortized by their parallel execution, the impact of alternative task implementations can be expected to produce the greatest measurable effect when applied to program sections of greater computational intensity. To examine the impact most effectively, sample runs of problem sizes up to 20^3 were profiled to determine the most suitable part of the computation to target first.

The LULESH code is hierarchically structured, with a clear mapping to the description of its phases given by Karlin *et al.* [5]. Its straightforward call graph permits a run time profile to be obtained using only back-to-back timings that separate regions featuring explicit computation, and calls to further functions. Starting from a timing of an iteration of the main program loop, repeatedly refining a profile of greatest part indicated that the function CalcFBHourglassForceForElems contributed the greatest amount of explicit computations. That conclusion was confirmed by finding that it accounted for 24.72% of execution time for a small problem size run with the *callgrind* profiler [6], so the function, which is part of the force calculation in the program's first stage[5], was selected as an initial target for testing.

Comparison of Explicit and Implicit Task Dispatch

The performance trade-off between slicer-generated tasks and explicit task specification was examined on the development system, as the conclusion became apparent already at small problem sizes. Total application execution times from a strong scaling test with 10^3 and 20^3 problem sizes are shown in Figure 1.

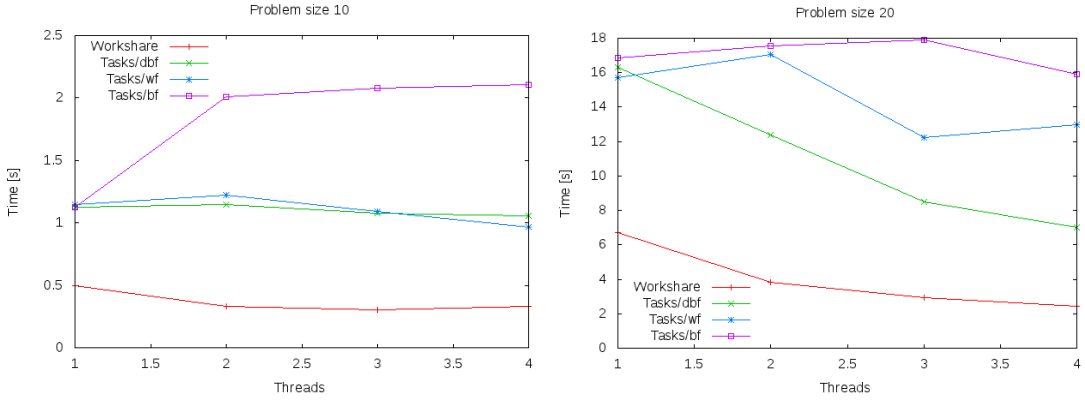

Figure 1: Run times of task-construction and scheduler interactions

The curves in Figure 1 are labelled with the scheduling policies they were run under, with *dbf*, *wf* and *bf* denoting *distributed breadth first*, *work-first*, and *breadth-first* scheduling policies, as described by Duran *et al.* [7]. The *Workshare* curve represents implicit task creation from work-sharing construct, which was not observed to behave differently with variable schedulers. While it should be noted that increasing problem sizes do produce improvements in strong scaling for the explicit task approach, the absolute overhead of the approach produces run times one order of magnitude greater than implicit task creation, making the latter the obvious implementation choice for performance testing.

Performance Results

Performance measurements of full application runs were collected on a compute server featuring dual 8-core Xeon E5-2670 processors, clocked at 2.6 GHz, and with a memory bus clock at 1.6 GHz. This system is not an operational supercomputer node, but is purposefully designed to resemble a node in an SGI ICE X system, to admit research outside the constraints of a production environment.

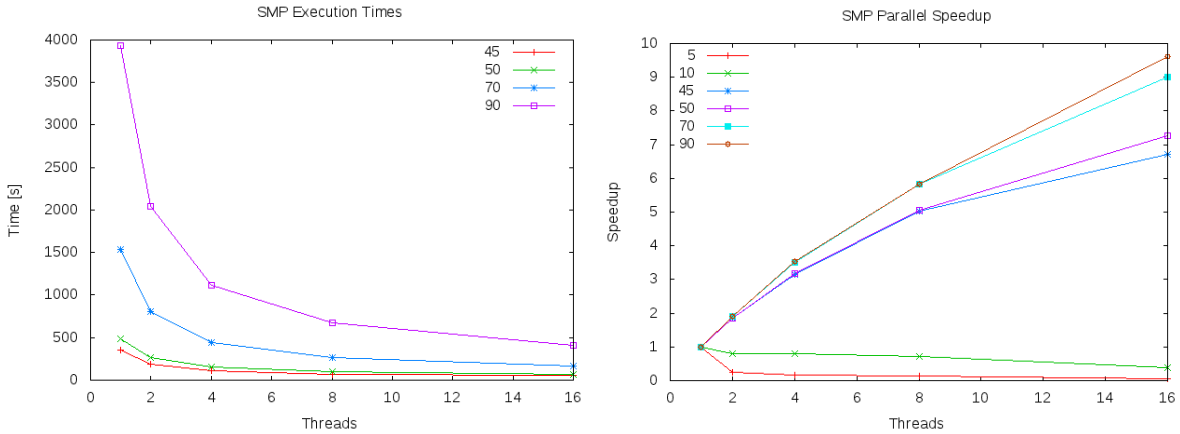

Figure 2: Run times and parallel speedup on 16-core system for all problem sizes

Figure 2 shows the recorded run times and parallel speedup curves obtained running LULESH with OmpSs. SMP execution times for problem sizes 5 and 10 are omitted, as these problems were too small to give visible results on the same scale as the remaining problem sizes, or yield any parallel speedup. Per-iteration timings of the hourglass computation were recorded for 16-thread runs, for use in the analysis of hybrid execution.

Hybrid Case Study

OmpSs supports both dispatching task functions to CUDA-enabled graphics accelerators, and a *versioning* scheduler module that is aware of multiple implementations of the same task, as described by Planas *et al* [8].

This section describes experiments intended to determine how this this mechanism may apply to LULESH.

CUDA support in the applied version of the OmpSs environment regrettably proved to have issues with the version of the Nvidia compilation tools and OS driver module on the test system, which made it necessary to utilize a separate system for this purpose, to allow for altering the operating system configuration. The system which was used for this purpose features an older GeForce 8800 GPU, which unfortunately limits the OmpSs run time from executing GPU-targeted and SMP tasks concurrently, and showed issues with larger data transfers. Because these technical restrictions made it unsuitable to accurately test properties of the full application code, this section presents a modeled estimate of program characteristics, and includes empirical observations to the extent that they could be obtained.

Application Porting

To assess the programming aspect, a minor parallelized loop in the program initialization stage was rewritten as a task function targeting GPU devices, and ran successfully in conjunction with the remaining SMP-targeted tasks, producing verifiable results for problem sizes 5 and 10. This amounted to replacing the original code with a function call, annotating the function signature with inputs, outputs and an expression to evaluate their extents at run time, and implementing the loop body as an external CUDA C function in a separate source file.

The programmability of this approach is somewhat superior to direct implementation using CUDA, since it removes the need for separate memory management code addressing host and device architectures, and does not require the explicit dimensioning of grid and thread blocks which is required for a native CUDA kernel invocation. Its utility as a tool to unify multiple code bases is present, but in the case of LULESH, its utility is limited as the target device language remains transparently visible in the kernel codes. Since the overarching coordination code which invokes the computational kernels constitutes a relatively small portion of the code, maintenance of the per-architecture kernel sets would amount to separate instances of almost the entire code base, with a comparatively small shared loop code to invoke them. This balance of coordination logic and computational intensity may be more favorable for a full-scale application, but investigating the potential benefit to the ALE3D setting LULESH was extracted from is beyond the scope of this whitepaper.

Performance Results

Apart from its abstraction of the memory management details, OmpSs utilizes device kernel code verbatim, apart from how its formulation does not explicitly utilize knowledge of the CUDA grid block sizes, becuase the kernel dispatch is generated by the compiler. This makes it possible to project performance characteristics without producing a complete port of the code. Making a similar consideration of the need for computationally intensive tasks, this section will examine the hourglass force computation, as above.

Task Transfer Cost Estimation Method

To approximate run-time system's management of data transfers, a synthetic benchmark consisting of a computational kernel making a single array update was written and annotated in OmpSs, and tested on the functional installation. Because the dependency management may detect unused values and omit their transfer, a small benchmark was devised to similarly update the array in host memory to force its transmission, and run repeatedly with variable array sizes. For comparison, an identical benchmark was developed natively in CUDA.

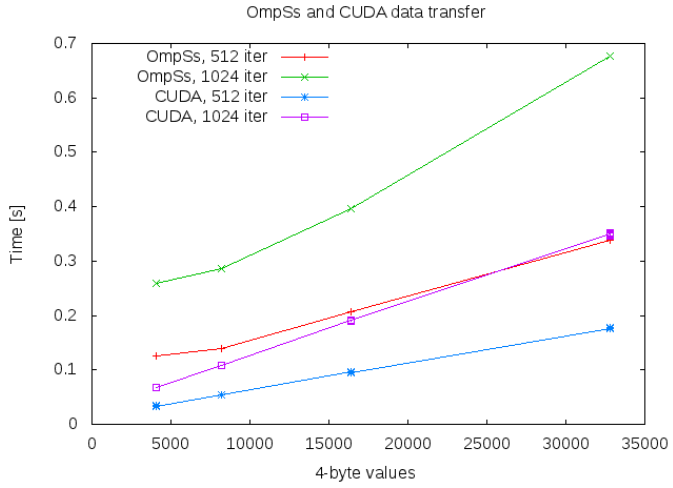

Figure 3: Comparison of data transfer benchmarks for OmpSs and CUDA

Figure 3 shows empirical timings of running these benchmarks for 512 and 1024 repetitions, which produced a large enough transfer cost to measure reliably. As the figure shows, the timings of the CUDA benchmark are in an approximately linear relationship with the OmpSs version as transfer sizes grow. Observing that the run-time system does introduce a noticeable overhead, we will treat data transfer costs as proportional to timings of CUDA benchmarks, with the reservation that it makes an optimistic cost estimate.

Computation Cost

The structure of the LULESH source code mentioned earlier is followed closely also in the existing pure CUDA port of the code, so an estimate of the computational cost of the hourglass computation can be obtained by instrumenting and running the CUDA version. Because computation kernel code passed to the Mercurium compiler is fundamentally already stated in CUDA C, this can be taken as an estimate of kernel performance when memory management is disregarded. Figure 4 includes timings of the hourglass force computation from the LULESH CUDA port, measured on a NVIDIA Tesla K20 GPU, in comparison with per-iteration costs recorded from 16-thread SMP runs. This class of GPUs was used as an approximation of the class of devices that may be expected to be coupled with nodes like the 16-thread SMP test platform, as they are present in the Abel Tier-1 resource which has nodes featuring dual Xeon E5-2609 CPUs, clocked at 2.4 GHz.

Task Transfer Cost Estimate

Data manipulation in both the OpenMP and CUDA ports of LULESH revolve around manipulating arrays that are collected in a single, shared data structure, representing the problem domain. For the OpenMP port, this structure resides in shared memory, while in the CUDA port, the hourglass computation is performed on a copy of it which resides in device memory. Only a subset of its contents are used by each computational kernel, so to estimate the cost of transferring hourglass computation as a task, its input parameters were identified to be 2 sets of values sized by the domain's number of grid elements and 3 sets sized by its number of mesh nodes. Output values were similarly found to be 3 sets sized by the number of mesh nodes; remaining data use is local to the functions. Using these figures, the mesh data sizes of benchmarked problem instances could be used to adapting the CUDA data transfer benchmark as a test for these particular data sizes, as an estimate of migration costs for a task version of the hourglass computation.

Performance Potential for Hybrid Execution

Figure 4 shows a comparison of the measured computational intensity and data movement costs for the candidate task.

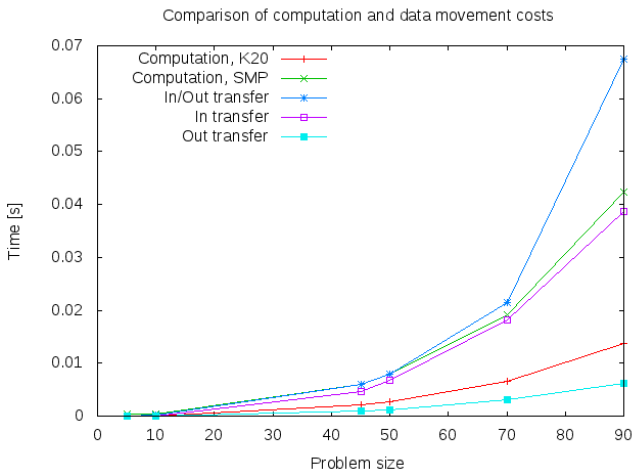

Figure 4: Comparison of estimated computation and data movement costs for a task version of hourglass force computation

As seen in the figure, the SMP computation cost of this kernel is similar to the cost of migrate the input values it depends on, and higher than moving the output values it produces, suggesting that it is not inherently infeasible to migrate its workload between SMP and CUDA devices at run time without the data transfer cost overshadowing all benefits to efficiency.

Having made this observation, there are several concerns which have not been addressed by this modeling effort. Specifically, data transfer cost is instrumented without run time layer overheads, the possibility to amortize one-time data movement cost over several iterations was not examined, and the kernel chosen for inspection contains the greatest computational intensity of all the candidates. The cost of migrating both input and output data is greater than the per-iteration SMP computation cost, suggesting that utilizing the superior computation speed of the GPU unit will result in a higher overall execution cost if all data dependencies are copied each iteration. If a part of the domain data structure could be relocated for several iterations, an overall advantage may be attainable, but this would require a significant revision of the problem domain data structure. Moreover, the majority of parallel work in LULESH remains far more fine-grained than the examined hourglass computation, suggesting that implementing the application in the form of schedulable, loosely coupled tasks would require a major redesign in order to produce a competitive implementation.

Conclusions

We have presented the results from a study to evaluate the interactions of the LULESH proxy application and the OmpSs programming model in a hybrid setting. Characteristics of an SMP implementation were shown, and the behaviour of a task-adaptation of the most computationally intensive step in the application was modelled with a combination of application instrumentation and benchmarking. While the modelled candidate task shows a certain potential for effective task-based implementation, the fine parallel granularity found in LULESH probably does not make it well-suited for implementation with a task-based approach in general, as computation is tightly synchronized. Performance measurements of a version which uses implicit tasking on SMP architectures admitted testing of design trade-offs with respect to the OmpSs run time system, and showed favourable scaling characteristics at the single node scale appropriate for shared-memory implementations.

References

[1] M. Lysaght, B. Lindi, V. Vondrak, J. Donners, M. Tajchman, PRACE-3IP D7.2.1 *A Report on the Survey of HPC Tools and Techniques*, http://www.prace-project.eu/IMG/pdf/d7.2.1.pdf

[2] Programming Models @ BSC, http://pm.bsc.es/

[3] The OpenMP API specification for parallel programming, http://www.openmp.org

[4] Ian Karlin, Abhinav. Bhatele, Bradford L.. Chamberlain, Jonathan. Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang and Daniel Wong: *LULESH Programming Model and Performance Ports Overview*, Technical Report LLNL-TR-608824, Livermore, CA, 2012.

[5] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, Charles H. Still: *Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application*, proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing

[6] Valgrind, http://valgrind.org

[7] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé: *Evaluation of OpenMP Task Scheduling Strategies*, proceedings of WOMP'08 Proceedings of the 4th international conference on OpenMP in a new era of parallelism.

[8] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta: *Self-adaptive OmpSs Tasks in Heterogeneous Environments*, proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing

Acknowledgements

This work was financially supported by the PRACE-3IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.