Performance Scalability on Embedded Many-Core Processors

Michael Champigny mchampig@mc.com
Mercury Computer Systems, Advanced Computing Solutions

Introduction

Shared memory, coherent embedded microprocessors such as the Tilera TILEPro processor family continue to scale the number of processing cores available with each generation. As we enter an era of many-core embedded computing devices the challenges to software developers continue to grow.

Mapping algorithms and applications to 10s of cores today to 100s of cores tomorrow requires performance scalability that is transparent across a range of computer architectures. These architectures often exhibit variable communication and compute costs as found in commodity cc-NUMA chip multiprocessors (CMP). The memory hierarchies may vary in their block size, capacity, number of levels, and coherency. The processing elements may be asymmetric, differing across a single die in clock speed due to thermal conditions, process variation, multiple voltage domains, and feature sets. The algorithms may contain data dependencies or may be composed in different ways which results in additional variability of computation and communication at runtime. Such variances reduce the effectiveness of static partitioning and require sophisticated load balancing strategies that account for the state of the runtime and available parallelism among other factors.

One strategy for achieving performance scalability is to employ *cache-oblivious* algorithms. These algorithms recursively decompose the input data set into units of work, called *tasks*, until the data size associated with leaf tasks can fit into the closest memory to a processing element. This provides a measure of locality that may be exploited at runtime with a dynamic scheduling policy.

A popular technique for efficient dynamic scheduling of cache-oblivious algorithms on a chip multiprocessor is *work stealing* [1, 2]. Work stealing schedulers attempt to keep all available processors busy with useful work through load balancing and employ various heuristics to ensure that caches are used effectively and memory growth is bounded at runtime.

We describe the design of a dynamic scheduler which uses a variant of the work stealing policy. We measure the performance scalability and parallel efficiency of several industry benchmarks taken from MIT Cilk, SPLASH-2, PARSEC, and Barcelona OpenMP. These benchmarks employ a task-based programming model where fine-grained tasks are exposed to a runtime scheduler responsible for mapping the available parallelism to the logical processing elements. Source code annotations similar to those available in Cilk and OpenMP are used to expose parallelism. These annotations are implemented using the standard C preprocessor for broader portability.

We demonstrate scalability to large core counts, in our case 64, on the Tilera TILEPro64 processor for a selection of numerical benchmarks without tuning algorithm parameters favoring a specific cache size or configuration. Our goal is to demonstrate the efficacy of cache-oblivious algorithms for performance scalability. As a baseline we compare the scalability against a commodity multi-core processor: the Intel i7-920 Quad Core Nehalem with HyperThreading.

Dynamic Scheduling

A CMP in the near future will feature 100s to 1000s of cores with complex memory hierarchies. These processors will have a unified, coherent address space but with a much higher cost for remote memory accesses in a NUMA configuration. The DARPA UHPC program [3] determined that the language and runtime layers of the software stack of massively parallel systems must have a mechanism of expressing and exploiting fine-grained parallelism in the form of tasks. To that end, dynamic scheduling has become a critical piece of software infrastructure and has a direct impact on the scalability and performance of current and future workloads.

Our parallel programming platform is composed of a small set of language annotations built on top of ISO C++, a runtime scheduler and execution model, and support for various programming models. In our scalability study we utilize the task parallel programming model in several industry standard benchmarks for shared memory computers. We determine where the major scalability bottlenecks are and address them directly with architectural and implementation changes in the runtime scheduler. We focus initially on the scalability of numerical algorithms for an embedded many-core processor, the Tilera TILEPro64. We verified that any design decisions made in the runtime to accommodate the TILEPro64 would benefit or at least not affect the scalability of the runtime on other platforms.

Work stealing schedulers operate by executing locally available tasks on each logical processor in depth-first order to mirror a sequential execution. When local work is exhausted, the local processor enters the idle state and looks for another logical processor, called the *victim*, to take tasks from. This operation is called *stealing* and the perpetrator is labeled a *thief*. The data structures in a work stealing scheduler must be designed to ensure that operations on it are free of hazards but also efficient to reduce unnecessary contention and overhead. In addition, several heuristics are used to encourage idle logical processors to look for large amounts of work to steal to amortize the cost of this expensive operation.

Most current work stealing schedulers choose a victim with a uniform random distribution. At scale, this policy results in poor locality and excessive cache misses. Instead, we group logical processors into a hierarchy of NUMA-like nodes. These logical NUMA nodes may or may not correspond to the physical topology of the processors. However, in our experiments we noted improved scalability over randomized victim selection even on UMA processors.

We use a highly configurable distributed data structure for holding ready tasks in the runtime scheduler which facilitates rapid experimentation with various scheduling policies. Each logical processor is associated with an instance of this data structure. The simplest structure is a linear list of tasks. At up to 32 cores, this structure scaled well but idle logical processors were limited to stealing a single task at a time. Since the frequency of steal operations is proportional to the amount of computational imbalance, highly irregular workloads suffer from contention and excessive overhead. To further amortize the cost of stealing, we group tasks into blocks each holding a fixed number of tasks. The resulting steal operation transfers several tasks from a victim to the thief. While this reduced contention, some workloads include phases where relatively few tasks are available in the system and therefore parallelism is limited. When blocks are a fixed size, load imbalance increases and we observe poor scalability above 48 cores. We make the observation that the amount of tasks to steal should be proportional to the number of available tasks in the system, and our data structure is modified to store a set of trees that grow logarithmically with the number of tasks in the system. The steal operation on this arrangement transfers a fraction of available tasks from the victim to the thief. Using this structure, we were able to scale to all available cores on the TILEPro64 processor with no observed penalty on a commodity multi-core processor.

Performance Scalability Benchmarks

We measure scalability on as many as 57 tiles on the Tilera TILEPro64 and show the speedup at several reference points corresponding to common processor configurations. Preliminary benchmark results are shown in Figures 1 and 2. These benchmarks were adapted from the MIT Cilk distribution and represent cache-oblivious versions of numerical algorithms. In both cases, minimal changes were made to the benchmarks. The Cilk keywords were replaced with our own annotations, but otherwise the algorithmic structure was unchanged. Because these algorithms are cache-oblivious, they subdivide computations until the terminal computations can fit into any data cache. The algorithms can therefore scale across a range of devices, from commodity multi-cores to embedded many-cores.

We demonstrate the results on at least two processor platform configurations to show that a common runtime scheduler can make effective use of a range of hardware. The first is an embedded many-core device, the Tilera TILEPro64 which represents a UMA configuration with 64 32-bit tiles connected via the iMesh interconnect. The second is a popular multi-core device, the Intel Core i7-920 which represents a cc-NUMA configuration with 4 64-bit cores connected via the QPI interconnect. Simultaneous multi-threading (SMT) via HyperThreading technology provides for an additional 4 logical processors.

Figure 1 shows the scalability results of a single precision heat diffusion algorithm employing Jacobi iterations on a matrix of size 4K-by-0.5K. Figure 2 shows the scalability results of a single precision LU decomposition with a matrix size of 1K-by-1K.

Summary

We described various strategies to improve work-stealing schedulers to promote locality and reduce overhead during the load balancing operation on embedded many-core processors including an improved victim selection strategy that considers locality as well as aggregate stealing of tasks in proportion to the amount of available work in the system. More importantly, we observed that the commonly employed randomized victim selection policy scales poorly at high core counts.

Our preliminary benchmark results demonstrate the scalability of our runtime scheduler across all core counts on task-based, cache-oblivious numerical algorithms. That is, we observe a performance benefit from all available processing elements on the TILEPro64.

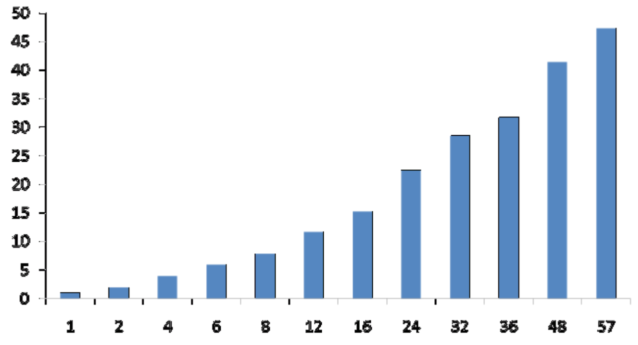

Figure 1: Heat Diffusion TILEPro64 Scalability.

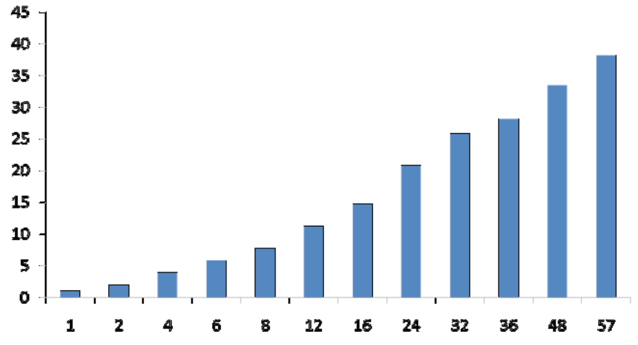

Figure 1: LU Decomposition TILEPro64 Scalability.

References

[1] G. Cong et al., “Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing,” *ICPP’08: Proc. Parallel Processing*, IEEE-CS, pp. 536-545, 2008.

[2] J. Dinan et al., “Scalable Work Stealing,” *SC’09: Proc. Conference on High Performance Networking and Computing,* ACM, pp. 1-11, 2009.

[3] W. Harrod, “Ubiquitous High Performance Computing Program”, DARPA-BAA-10-37, 2010.