ExaScience Lab
Intel Labs Europe

EXASCALE COMPUTING

Improving the Data Locality of Work Stealing - A Domain-specific Approach

P. Costanza
B. De Fraine
T. Van Cutsem

Report 07.2011.1, July 2011

Improving the Data Locality of Work Stealing

A Domain-specific Approach

Pascal Costanza   Bruno De Fraine   Tom Van Cutsem

Vrije Universiteit Brussel, Software Languages Lab
pcostanz,bdefrain,tvcutsem@vub.ac.be

Abstract

Fork/Join parallelism based on work stealing is becoming a widely used approach for parallelizing programs, yielding good and proven performance characteristics, while being relatively convenient to use. A known problem, however, is that randomized work stealing may suffer from suboptimal data locality over time. Current solutions for this issue rely on programmers defining worker affinity for tasks, or defining locales or places where tasks can be explicitly placed when they are generated. They thus add cognitive overhead. In this position paper, we propose to complement these existing solutions with a domain-specific approach: Based on the observation that most parallel algorithms can be classified according to "dwarfs", we suggest to include explicit support for such classes of algorithms on top of the basic Fork/Join operations. Such additional support (in terms of language constructs or library extensions) can improve data locality based on domain-specific knowledge such as how a specific dwarf is typically realized, and especially what its typical data access patterns are. We illustrate our idea with structured grid methods, one of the well-known dwarfs.

1. Introduction

The fork/join programming model and the work-stealing scheduler are increasingly popular techniques for parallel programming, as exemplified by approaches such as Cilk (spawn/sync) [4], the Java Fork/Join framework [8], X10 and Habanero Java (async/finish) [7] and Intel Threading Building Blocks. Fork/join is a light-weight task parallelism model where computations are dynamically created and a runtime scheduler is responsible for scheduling the computations across the workers. Work-stealing algorithms organize such runtime scheduling in a distributed manner by storing the tasks for each worker in a local deque. Busy workers will push and pop tasks from the bottom of their deque using synchronization-free operations. When a worker becomes idle, it will try to *steal* a task from the top of another random busy worker's deque. Work-stealing scheduling achieves time, space and communication bounds that are all within a small constant factor of optimal [4].

A known issue of randomized work stealing is that it may suffer from suboptimal data locality over time, when tasks being generated for some part of a data structure in one iteration of an algorithm do not get stolen by the same workers that already executed tasks on the same part of the data structure in previous iterations. This problem has been described by Acar et al. [1], who refer to such algorithms as *iterative data-parallel applications*. Other cases of data locality issues with work stealing have, for example, been reported by Lea [8].

Acar et al. [1] suggest to improve data locality of work stealing by allowing tasks to be associated explicitly with workers. This is enabled by adding a mailbox to each worker from which it can fetch tasks before attempting to steal tasks from other workers. Tasks can then be placed explicitly in such mailboxes in addition to queueing them locally, thus improving chances that they are executed by the corresponding worker.[1] A downside of this approach is that, by default, it relies on programmers giving explicit hints in their application code when spawning tasks, and thus adds cognitive overhead when parallelizing applications. This is especially problematic in environments where the loads and speeds of processors is unkown and may change over time. For example, such applications may have to execute in heterogeneous hardware environments with different kinds of processors, and/or may run on desktop machines where several applications compete for processing time on the available cores.

In this position paper, we propose to complement the solution suggested by Acar et al. with a domain-specific approach in order not to burden application programmers with the requirement to give such explicit hints. Based on

---

[1] This explicit affinity of tasks to workers is somewhat similar to explicit *places* or *locales* when spawning tasks in Chapel [5], SLAW [7] or X10 [6], although we are not aware of any implementation of the latter approaches that allows for stealing tasks across places or locales.

2010/9/13

```
procedure timestep (grid, function)
  if there is only one row in the grid
  then update each element in the row sequentially,
         using the function;
  else divide the grid into two halves of rows;
       recursively fork timestep on each half;
       join;

procedure mapgrid (grid, function, n)
  repeat the following for n iterations
    timestep(grid, function);
```

Figure 1. The non-adaptive version of mapgrid.

the observation that most parallel algorithms can be classified according to "dwarfs" [3], we suggest to include explicit support for such classes of algorithms on top of the basic fork/join operations. Such additional support (in terms of language constructs or library extensions) can then take domain-specific knowledge into account how a specific dwarf is typically realized, and especially what its typical data access patterns are. In the following, we illustrate our idea using the "Structured Grids" dwarf, because it is a simple, but non-trivial example for this purpose.

2. Structured grids

On the wiki page related to their effort, Asanovic et al. described the "Sructured Grids" dwarf as follows [2]:

> Data is arranged in a regular multidimensional grid (most commonly 2D or 3D). Computation proceeds as a sequence of grid update steps. At each step, all points are updated using values from a small neighborhood around each point. The general form of a structured grid computation is as follows:
> $$\forall \vec{i} \in \text{Indices} : d'[\vec{i}] = f(\{d[\vec{i}+\vec{\delta}] : \vec{\delta} \in \text{Neigborhood}\})$$

An example is the Jacobi method for solving Poisson's equation (which arises in heat flow, electrostatics, gravity, etc.). Another example is John Conway's Game of Life. A structured grid method is characterized by its *stencil* (which specifies the neighboring values that are used to update a grid point) and the order of updates.

Structured grids allow a high degree of parallelism when the grid points being updated are not used as neighbors in the same step (for example, because the new values are stored in a different version of the grid instead of being updated in place). Each processor is usually assigned a contiguous subgrid, and can perform each update step locally. It only has to communicate and synchronize with neighboring nodes.

A procedure mapgrid can serve as a domain-specific abstraction for performing a structured grid computation. Fig. 1 shows a straightforward implementation of mapgrid in terms of fork/join (in pseudocode). It takes a grid, a function and a number of iterations as parameters. For each iteration, the procedure timestep is called, which applies the

```
procedure heatEquation (grid, n)
  function new-value (old-value)
    new-value is old-value +
      nu * (east(old-value) + north(old-value) +
            west(old-value) + south(old-value)
            - 4 * old-value)

  mapgrid(grid, new-value, n);
```

Figure 2. The heat equation using mapgrid.

```
procedure timestep (grid, function)
  if there is only one row in the grid
  then update each element in the row sequentially,
         using the function;
       increment the current worker's work counter;
  else divide the grid into two halves of rows;
       recursively fork timestep on each half;
       join;

procedure mapgrid (grid, function, n)
  distribute the grid's rows over the workers;
  repeat the following for n iterations
    fork timestep on each worker;
    join;
    redistribute the rows over the workers,
    taking the workers' work counters into account;
```

Figure 3. The adaptive version of mapgrid.

passed function to each grid point to calculate its new value for the next time step. The procedure timestep divides the grid into subsequences of rows on which timestep is recursively forked, until a single row can be processed sequentially.[2]

Fig. 2 shows how the heat equation can be expressed in terms of mapgrid. As mentioned above, this use of fork/join suffers from suboptimal data locality, because each worker will work on different parts of the grid in each time step.

Fig. 3 shows our proposal for an alternative implementation of mapgrid where the rows of the grid are distributed over the available workers in each iteration. We assume that tasks can be explicitly placed on specific workers, as in the approach by Acar et al. [1]. Initially, this leads to an even distribution of the rows over the workers. The procedure timestep is essentially the same as in Fig. 1, but additionally, our new version counts how much work is performed, i.e., how many rows are processed, *per worker*: Even tasks that got stolen by a worker will increment the thief's work counter, not that of the worker that originally spawned the stolen task. These work counters eventually allow mapgrid to redistribute the rows after each iteration using the *actual* work done by the workers.

---

[2] Other strategies for dividing the grid could be used here as well, for example using quadtrees.
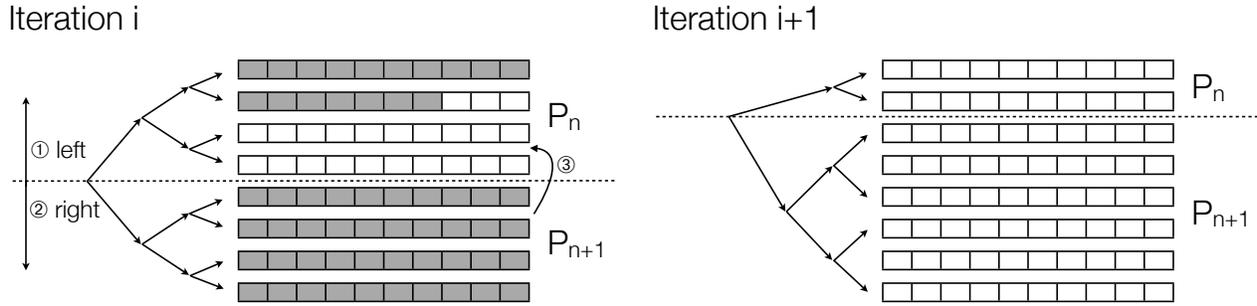
Figure 4. Two time steps in the adaptive grid iteration.

Fig. 4 illustrates two subsequent time steps in our adaptive implementation of `mapgrid`: Each processor first forks a task for the left half of the passed grid (1), and then for the right half (2). If in iteration i a processer P_{n+1} is faster at processing rows than others, it will steal rows from other processors (3), and thus have a larger work counter than other processors. In the next iteration $i + 1$, processor P_{n+1} will therefore get a larger portion of the grid from the start. Instead of stealing randomly after finishing its own portion of the grid, we suggest that a procesor *first* attempts to steal from the processor working on rows to the left of the currently assigned rows (3), in order to further improve data locality both for the current and for subsequent time steps.

3. Discussion and future work

Note that applications of `mapgrid` (like the one in Fig. 2) do not have to change in order to take advantage of our improved implementation of `mapgrid`. We can take advantage of specifying worker affinity, as in [1], but are able to abstract this away because `mapgrid` is effectively a domain-specific abstraction for performing structured grid computations that can be optimized based on knowledge about the regularity of data accesses in such computations. This abstraction relieves the application programmer from having to provide locality hints, but the flip side is that he or she is restricted to the intended computation pattern, from which it may be hard to deviate. It may be possible to provide some middle ground here, for example by providing extension points to the programmer.

So far, we have restricted our efforts to the case of structured grid algorithms, which admittedly have very regular and predictable data access patterns. We intend to investigate how our domain-specific work-stealing approach can be extended to more complex settings (such as structured grids with adaptive mesh refinement) and to other dwarfs (such as N-body methods or particle-in-cell codes). We expect that this will require different abstractions, which – given the increased complexity – may be harder to identify, but which may also offer greater benefits.

Our approach is currently in a prototyping stage, which prevents us from collecting performance results that allow apt comparisons with approaches that rely on manual specification of locality hints. We plan to incorporate our ideas in one of the existing fork/join frameworks.

Acknowledgments

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Symp. on Parallel Algorithms and Architectures (SPAA'00)*. ACM, 2000.

[2] K. Asanovic, K. Yelick, J. Shalf, and R. Bodik. Structured grids. Wiki page at `http://view.eecs.berkeley.edu/wiki/Structured_Grids`, Feb. 2008.

[3] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 2005.

[7] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In *Int. Parallel and Distributed Processing Symp. (IPDPS 2010)*. IEEE, 2010.

[8] D. Lea. A java fork/join framework. In *JAVA'00*, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3.