Towards scalable implicit communication and synchronization

Explicit concurrency control, unscoped synchronization and implicit data dependencies considered harmful

R.C. Poss C.R. Jesshope
Universiteit van Amsterdam
{r.c.poss,c.r.jesshope}@uva.nl

Abstract

There exists several divides between implicit and explicit paradigms in concurrent programming models, for example between the assumption of coherent shared memory (e.g. OpenMP), and the assumption of distributed memory (e.g. MPI). Explicit paradigms exist to provide control to programmers, but cause scalability concerns: programs need to be adapted whenever the granularity of concurrency changes. With the rise of large heterogeneous pools of computing resources, we must increasingly distribute tasks automatically. Implicit paradigms allow this in theory and are desirable for expressivity and intuitiveness, but their scalability in heterogeneous environments is yet unclear. In this position paper, we propose to consolidate previous knowledge by seeking more implicit concurrent programming models that combine three properties. The first desirable property is *resource agnosticism*, where programs separate clearly the description of computations from the description of task distribution to resources. The second property is *scoped synchronization*, where programs express no more synchronization than required by the described computation. The third property is the *visibility of data dependencies* between tasks by compilers and run-time systems. Only when these properties exist together, it becomes possible to automatically tailor programs to heterogeneous target systems and achieve efficient execution. We show how *specializability* is needed to optimize this process.

Categories and Subject Descriptors D.1.3 [*Concurrent programming*]: Distributed programming; D.3.2 [*Language classifications*]: Concurrent, distributed, and parallel languages; D.3.4 [*Processors*]: Run-time environments

General Terms Distributability, visible data dependencies, resource agnosticism, scoped synchronization, specializability.

Keywords Heterogeneous concurrency resources, concurrency granularity, concurrent programming models, dataflow programming, cloud computing, concurrency mapping and scheduling

1. Introduction

We identify the following three distinctions in current concurrent programming models. A first distinction exists between implicit and explicit *concurrency control*, i.e. the management of the existence of tasks and the mapping and scheduling of tasks to execution units. Another distinction exists between implicit and explicit *communication*, mostly between models assuming shared memory and those assuming distributed memory. The third distinction exists between implicit and explicit *synchronization*, i.e. the negociation of "rendezvous" points between asynchronous tasks.

Programming paradigms have long favored side effects and implicit communication for practical reasons. As early as 1985, the author of [18] highlights that CSP, where explicit message passing finds its roots,

> strongly discourages the programmer from carving his program into a large number of processes. This psychological disincentive operates whether or not there is any underlying difference in efficiency of nonlocal versus local accesses. [...] thus parallelism at small levels of granularity is discouraged. Large processes require large processors and thus lead inexorably toward systems with relatively small numbers of large processors and away from "myriaprocessors"—highly parallel machines with relatively small processing elements—that might exploit coming technology more effectively.

Despite notable efforts at implementing fine-grained CSP-based models efficiently over large parallel systems, such as Occam and the transputers [11, 23], these efforts have not been repeated recently. Besides, we observe that explicit concurrency causes new scalability concerns: explicit programs need to be adapted whenever the granularity of concurrency changes, and support for heterogeneous environments is difficult. This is a significant issue today, because massively concurrent execution resources are becoming ubiquitous but available as heterogeneous pools. Manual tailoring of parallelism to many different resource characteristics becomes simply too expensive. We postulate the following:

1. explicit concurrency control exist only to provide control of resources to programmers by lack of better, automated solutions via compilers and run-time systems; and
2. expliciteness makes reasoning about programs difficult, and restricts the automated transformation of programs to the specific properties of concurrent resources.

Using the words and the spirit of [9], we suggest that explicit concurrency control, global synchronization and unstructured data dependencies should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, system-oriented languages where run-time systems are implemented). We propose to move in this direction by seeking the following three general properties. The first desirable property is *resource agnosticism*,

AMP '10 June 6, 2010, Toronto, Canada

where programming environments separate clearly the description of computations from the description of mappings of tasks to resources and the scheduling on execution units. The second property is *scoped synchronization*, where programs express no more synchronization than required by the computation. This can be achieved either by implicit synchronization or scoped explicit synchronization. The last desirable property is the *visibility of data dependencies* between tasks to compilers and run-time systems, for instance by providing explicit dependencies or structured implicit dependencies in programming languages. We explain and illustrate these properties in the following sections.

These three properties already exist in many current models, but are rarely found together. We show that when they are available, programs can be both *distributable* and *specializable* to specific target architectures either at compile-time or at run-time. Scalability in heterogeneous environments can then be achieved without changes to programs.

2. Resource agnosticism

Resource agnosticism is present in a programming model when programs *expose* concurrency in computations but do not explicitly *manage* it. Concurrency is exposed when programs indicate when operations can be executed concurrently and/or when they must be executed in sequence. Management occurs when programs define the distinction between *tasks* (where computations are described) and *execution threads* (where execution takes place), define the existence of threads, or define how and when to map and schedule tasks onto threads. By definition, resource agnosticism mostly opposes explicit concurrency control.

We can recognize that resource agnosticism exists when the programming model does not expose to programmers which sequential processor is in charge of specific parts of the computation, nor provides handles to control threads and the distribution of tasks. When looking at the POSIX threads interface [19] and the corresponding programming model, we don't see much resource agnosticism. Programs must control threads and the mapping of tasks explicitly. Thread management overheads mandate a coarse-grained exposition of concurrency. Instead, OpenMP [25] provides much more resource agnosticism. The most common use of OpenMP is to *annotate* loops to indicate how they can be run concurrently. The responsibility of the mapping and scheduling of tasks to threads is delegated to the OpenMP run-time system, and so is the creation and termination of execution threads. Of course, pure dataflow programming is the ultimate form of resource agnosticism. Many models derived from dataflow concepts including Cilk [4] (functional concurrency), S-NET [15, 17] (streams) and SAC [14] (both functional concurrency and data parallelism) are mostly resource agnostic, since they suggest to programmers to define only workload independence and typically do not offer controls on task distribution. Of course, resource agnosticism does not exclude explicit concurrency control entirely, as long as it stays *external* to the description of computations. We highlight that explicitly distributed programs (e.g. with MPI) are mostly not resource agnostic however. Indeed, they assume strict process boundaries, from the assumption of resource boundaries. This assumption limits optimization opportunities when coalescing processes on common nodes, because process boundaries must be enforced [27] to preserve program semantics.

3. Scoped synchronization

In the general sense, synchronization occurs in two situations. The first is *precedence synchronization*, when progress in some task(s) is dependent on progress in some other task(s). The other form of synchronization is *exclusion*, necessary when sharing state between independent tasks. In both cases, multiple tasks engaged in synchronization must communicate synchronization events. When concurrency is mapped across space, this creates requirements on interconnects and networks. Scoping synchronization aims at limiting the scope of synchronization events, in order to provide both the opportunity to run-time systems to map programs to hardware that satisfies the requirements efficiently, and a bound to the cost of the communication of events.

Here we address implicit and explicit synchronization separately. With explicit synchronization, programming models offer conceptual devices that offer synchronization services, shared between two or more sequential processes. For example, a mutex lock is such a device for exclusion. Condition variables are a simple device for precedence synchronization, but so are thread *teams* in UPC [28] and *communicators* in MPI when involved in collective communication. With synchronization devices, scoped synchronization can be achieved if the environment can determine certainly, before tasks are distributed to resources, which tasks have access to which devices. This is required if the communication costs are to be optimized. Here, we observe that programming models that assume shared memory typically expose poor scoping of synchronization, because synchronization devices are assumed to exist in a global pool visible from every task (the shared memory) and visibilities are not explicitly declared. We identify Chapel [5], Fortress [1], S-NET and SVP [20] as exceptions. With Fortress and Chapel, synchronization devices are lexically scoped in programs, and synchronization scopes are thus statically known. In S-NET, the link between *synchrocells* and computations is explicit and visible prior to mappings. With SVP the only explicit synchronization device is the *exclusive place* which offer the "secretary" service described in [10]. Access to this device must be requested explicitly by tasks to a controller authority also in charge of distributing work; the controller thus knows exactly the scope of exclusion.

With implicit synchronization, scoping is usually available directly to the environment. For example when precedence synchronization is implicit via reads and writes to channels, the endpoints of the channels determine the scope. Scoping is also usually available with implicit synchronization on the termination of tasks, when the environment can determine, when tasks are created, which tasks can be dependent on their termination. This is usually the creating task, or possibly *continuation* tasks in models that have them (e.g. Cilk). Conversely, scoping can be lost if termination of an asynchronous task can be captured in a closure and stored and/or shared anonymously, like with "futures" in Multilisp [18] and "spawns" in X10 [6]. More generally, the presence and use of unscoped *barriers* in programs are a clear symptom of unscoped synchronization since they involve all tasks.

4. Visibility of data dependencies

Data-dependent tasks rely on communication services from the environment at run-time. For a given program, a given input and a set of characteristics for individual execution units, there exists a minimal set of requirements on the capacity, topology, and other properties of the communication links between tasks that maximizes efficiency at some level (e.g. performance vs. cost). We capture these requirements collectively as the *communication requirements* of programs. By definition these are not scalar values but functions of the input parameters and the execution resources.

The need to identify and describe communication requirements has long existed in HPC, where the cost of computing resources justifies detailed analyses of the behavior and properties of programs before the resources are acquired and the programs mapped. More recently, communication requirements have become essential to the pricing of services in cloud computing. In [3] and [2], the authors identify previous work in the classification of communication requirements across a range of computing areas. However,

to our knowledge, efforts to derive automatically communication requirements from programs and model their evolution across heterogeneous resource characteristics have been limited so far. Therefore, we propose to steer future work on programming models towards deducing communication requirements automatically from the data dependencies between tasks in programs. This in turns requires that data dependencies are *visible* to the environment *prior* to the mapping and scheduling of tasks to resources, and that they expose enough information to deduce communication requirements.

Generally, explicit communication (e.g. via message passing) can expose dependencies to automated tools when programs use high-level operations [13]. Issues start to arise with implicit communication, and were partially identified as early as [29]. A common situation is the assumption of near-uniform access to a shared memory, which allows programs to express arbitrary, uncoordinated data dependencies between any pair of tasks. The dependencies in such programs are typically not analyzable before the programs are executed, i.e. after mapping and scheduling have been decided. This situation is pervasive in models that allow and encourage the use of global shared state and side-effects, but also exists in more "controlled" models. In particular we highlight models that allow a task to allocate storage in a global pool and communicate only a *handle* to it to other tasks, while hiding the mapping of handles to storage from the environment. Although a data dependency on the handles is visible, the information needed for communication requirements is hidden between "producer" and "consumer" tasks that coordinate in this way. SVP and SAC are an example of this situation. We also point at the *consistency model* assumed by programs. A consistency model is a contract between programmers, compilers, and the environment that offers communication opportunities and restrictions between tasks. As such, well-specified weak consistency models [24] are desirable because they push programs to express more information on communication requirements in the data dependencies between tasks.

While this visibility property is more difficult to recognize in models offering implicit communication, we can provide a few examples. In S-NET, behavior is isolated in SISO and state-free *boxes* whose data dependencies stem from the network structure. With UPC, the data dependencies are partially visible through the partitioning of the global adress space and task *affinity* with portions of the address space. Dataflow-like programs can be written to use only explicit data dependencies, e.g. in Cilk or SVP. With Cilk++ [22], *hyperobjects* [12] allow implicit communication, but they are well-structured and can be extracted by the environment when they are scoped lexically in programs.

5. Distributability and specializability

The challenge of heterogeneous concurrency resources is the diversity of *concurrency granularities*. The concurrency granularity of a given execution environment reflects the threshold on the size of workloads where it becomes advantageous to expose them to the concurrency management system, as opposed to simply express them sequentially [26]. Ideal scalability over arbitrary concurrent resources requires programs to expose at least as much concurrency as available in resources, but the granularity of resources prevents programs from expressing "too much" concurrency due to overheads. A common assumption is that the granularity is homogeneous and known. In this context it is possible to tailor manually the expressed concurrency to the resources. With pools of heterogeneous resources with diverse granularities, possibly evolving over time, this assumption does not hold. We take cloud computing as an example, or large grids of clusters of multi-cores. The cost of manual adaptation then justifies seeking automated solutions to adapt programs to the granularity of the resources they are mapped onto.

Efficient automated solutions, if they exist, require two general properties from programs. The first property, *distributability*, is essential but intuitive. It requires that programs express concurrency so that it *can* be mapped onto effectively concurrent heterogeneous resources. This implies resource agnosticism, needed to allow a *diversity* of resources. It also requires visible scopes on synchronization, because mappings for program synchronization must be decidable over heterogeneous implementations of synchronization. Distributability is then increased when more concurrency is exposed *and* data dependencies are more visible, because knowledge of communication requirements is required to evaluate and decide mappings automatically in run-time systems.

Once distributability is available, programs can be mapped onto resources whose granularity is finer than the one expressed. The distributed S-NET system described in [16] is a successful example. We then highlight the situation with heterogeneous pools with both coarse and fine concurrency granularities in resources. Distributability suggests expressing more fine-grained concurrency in programs, in order to achieve effective distribution over the fine-grained resources. However while doing so, mappings should still be possible on coarse-grained resources. Here we propose the other desirable property from programs, that of *specializability*. This requires that in a program exposing fine-grained concurrency, any arbitrary *part* of the program can be be *transformed automatically* when mapped onto a coarse-grained resource to remove the overhead of managing the expressed concurrency.

We have not yet recognized explicit definitions and uses of specializability in current programming models, although the efforts described in [8, 27] suggest growing attention in this direction by the community. We do observe however that specializability is enabled in Cilk via the concept of *faithfulness* [22], which allows program parts to be trivially converted to sequential C code with identical semantics and no concurrency overhead. A similar opportunity exists in SVP, where any thread family can be transformed to a sequential loop.

Once both distributability and specializability are available in languages and used by programs, a run-time system can map any program to any assortment of concurrency granularities while providing efficient execution on each resource. The achieved concurrency can be augmented simply by expressing more concurrency in programs, without introducing overheads because the extra concurrency can be removed by specialization on coarse-grained resources. We predict that S-NET, when used in combination with a specializable box language, offers this opportunity fully. We have seen early attempts in this direction [7, 21], although object-oriented programming hides the scope of synchronization and dependencies and thus restricts mapping optimizations.

6. Conclusion

Most programming models today exhibit the three proposed properties—namely implicit concurrency control, scoped synchronization, visibility of data dependencies—to some extent. Not surprisingly, these properties are less present, even hindered, in concurrent programming models that evolved as extensions of sequential models (e.g. OpenMP). They are more pervasive with models dedicated to distributed programming (e.g. MPI), although these are still sensitive to granularity. However, we must seek also *specializability*, i.e. the ability to tailor programs parts automatically to multiple concurrency granularities. We point to S-NET in combination with Cilk or SVP as an example promising hybrid programming model that exposes this feature and enables run-time systems to optimize execution over increasingly complex sets of concurrency resources. We suggest that the general adoption of the proposed properties in future research will facilitate more scalable communication and synchronization across pools of heterogeneous resources.

Acknowledgments

The authors acknowledge the European Union for funding this work through the projects Apple-CORE (grant no. FP7-ICT-215216) and ADVANCE (grant no. FP7-ICT-248828).

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, March 2008. URL `http://research.sun.com/projects/plrg/fortress.pdf`.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[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. *SIGPLAN Not.*, 30(8):207–216, 1995. ISSN 0362-1340. DOI 10.1145/209937.209958.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. DOI 10.1177/1094342007078442.

[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 *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. DOI 10.1145/1094811.1094852.

[7] A. A. Chien, V. Karamcheti, and J. Plevyak. The Concert system – compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993. URL `http://www-csag.ucsd.edu/papers/concert-overview.ps`.

[8] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. DOI 10.1145/1065944.1065950.

[9] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. ISSN 0001-0782. DOI 10.1145/362929.362947.

[10] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971. ISSN 0001-5903. DOI 10.1007/BF00289519.

[11] J. Edwards and P. Lawson. The advances of Transputers and Occam. In J. Edwards, editor, *WoTUG-14 Occam and the Transputer-Current Developements*, pages 1–12. IOS Press, Amsterdam, 1991. ISBN 90-5199-063-4.

[12] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. DOI 10.1145/1583991.1584017.

[13] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004. ISSN 0164-0925. DOI 10.1145/963778.963780.

[14] C. Grelck and S.-B. Scholz. SAC: off-the-shelf support for data-parallelism on multicores. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 25–33, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. DOI 10.1145/1248648.1248654.

[15] C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(1):221–237, 2008.

[16] C. Grelck, J. Julku, and F. Penczek. S-Net for multi-memory multicores. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 25–34, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9. DOI 10.1145/1708046.1708054.

[17] C. Grelck, S.-B. Scholz, , and A. Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010. DOI 10.1007/s10766-009-0121-x.

[18] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. ISSN 0164-0925. DOI 10.1145/4472.4478.

[19] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[20] C. Jesshope. A model for the design and programming of multicores. *Advances in Parallel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008. URL `http://dare.uva.nl/record/288698`.

[21] V. Karamcheti and A. Chien. Concert-efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 598–607, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. DOI 10.1145/169627.169806.

[22] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. DOI 10.1145/1629911.1630048.

[23] D. May and R. Shepherd. Occam and the transputer. In *Advances in Petri Nets 1989*, volume 424 of *Lecture Notes in Computer Science*, pages 329–353. Springer Berlin / Heidelberg, 1990. ISBN 978-3-540-52494-6. DOI 10.1007/3-540-52494-0.

[24] D. Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993. ISSN 0163-5980. DOI 10.1145/160551.160553.

[25] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, 2008. URL `http://www.openmp.org/mp-documents/spec30.pdf`.

[26] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. *SIGARCH Comput. Archit. News*, 21(2):14–26, 1993. ISSN 0163-5964. DOI 10.1145/173682.165126.

[27] H. Tang, K. Shen, and T. Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Trans. Program. Lang. Syst.*, 22(4):673–700, 2000. ISSN 0164-0925. DOI 10.1145/363911.363920.

[28] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, May 2005.

[29] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, 1973. ISSN 0362-1340. DOI 10.1145/953353.953355.