On the compilation of a parallel language targeting the self-adaptive virtual processor
Bernard, T.A.M.

Citation for published version (APA):
Enschede: Print partners Ipskamp

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)
In this chapter, we present a selection of past and current work conducted in both academia and industry in parallel computing systems. We can thus evaluate the relevance of our research and position it against this work. Parallel computing has been researched since the 1970s where most of the concepts have been already defined [5]. At first, focused on scientific computing and HPC, its move to mainstream computing became clear in the 2000s [7]. We note that it is difficult to position our research since we tackle the multicore programming challenge with a full system approach introduced in Chapter [3]. Therefore, at this point, it is important, as illustrated in Figure [2.1] to distinguish the different domains and their influences. The comparisons with other work can be done at different levels of granularity of the system: at the model-level, comparing concurrency approaches; at the system-level, comparing the system as a whole against others; at the approach-level, in ways of dealing with concurrency; at the compiler-level, in the sense of how it deals with concurrency. This chapter answers these questions and also exposes at the end the requirements that we consider priorities for a concurrent execution model.
Chapter 2. Background in parallel computing systems

Figure 2.1: Overview of computing system domains and their intersections. The domain of execution models encapsulates the domains of programming languages, compilers and architectures. These domains are derived from the execution models with their properties. The compiler domain overlaps both domains of languages and architectures.

2.1 Approaches in concurrent execution models

Computing systems involving concurrency at some levels are already present in the market. With some exceptions, it is observed that approaches are made either at the architecture level or at the programming level. This section gathers an overview of execution models. It is important to note that this overview of approaches is not complete since new approaches appear frequently. Moreover, we note that there are different angles for observing each model. The programming model is the first aspect; then, each execution model targets a specific area which can be a particular architecture to program or more general-purpose targets. Furthermore, in the context of this thesis, we also look at the model’s implementation in their tool sets. Top-down approaches come from the software side and aim to target general architectures rather than a specific one. Bottom-up approaches project forces from the hardware layer onto the software layer; the resulting software definition is then dedicated for this architecture.

2.1.1 CUDA

A characteristic example of typical hardware and programming model currently available in the industry is NVidia’s Compute Unified Device Architecture [22] (CUDA). This execution model is pushed from the bottom-up by the architecture. In other words, the CUDA execution model is dedicated to CUDA graphics hardware and is not portable to other architectures. CUDA makes use of GPU arrays by providing a programming interface that allows the offloading of computations from the CPU via a high speed bus, for example AGP. The limitations of this model (both hardware and programming framework), when compared to the work in this thesis (SVP execution model in Chapter [3], are twofold: the programming model requires highly explicit concurrency through...
the use of many code directives, with the hardware itself relying on an ‘accelera-
tor’ model, where more general computations are carried out on a CPU, and hence computations carried out on the CUDA architecture are bound by bus speed limitations. Also, the scheduling of work/threads lacks the flexibility and concurrent composition allowed by the SVP execution model. The CUDA execution model defines concurrent regions where programmers do not need to manage thread creation or thread destruction. Moreover, CUDA employs Single-Program Multiple-Data (SPMD) techniques; a concurrent region will execute at run-time in a set of concurrent threads running at the same time.

2.1.2 Dataflow approach

Approaches with dataflow-like models have a long history of research which have now received additional focus, given the impending limits of conventional performance scaling. Lee et al. [23] present key points of dataflow architectures and multithreading. The dataflow model of execution offers attractive properties for parallel processing. This model captures program’s dependencies, is asynchronous, and bases instruction execution on operand availability; synchronization of parallel tasks is implicit in this model. Moreover, scheduling is dynamically achieved by satisfying program dependencies. Lee et al. also expose the convergence of the control-flow and dataflow models of execution; the authors expect to build hybrid architectures using dataflow approach in conventional control-flow thread execution. To conclude, this article says that the eventual success of dataflow computers depend on their programmability. In this thesis, the SVP execution model tackles this challenge with the Microgrid dataflow-driven architecture implementation (cf. Section 3.3) and the imperative-language μTC language implementation (cf. Section 3.4).

Historically, closely related work can be observed from around 20 years ago at MIT during Arvind’s research on combining dataflow principles with Von Neumann computing [24, 25]. This work explored the potential benefit of providing the scheduling of dataflow computing on conventional instruction sets. In Transactional Memory (TM) [26], dependencies are discovered dynamically, in contrast to the static capturing of dependencies in the SVP model which facilitates very fine-grained concurrency. TM can also be fine-grained as soon as there are not too many dependencies. More recent work similar to the principles used in SVP is the WaveScalar architecture [27]. This achieves pure-dataflow execution by using an implicit program counter for memory read/write ordering. However, there are restrictions in the amount of concurrency that this can expose: constraints on memory reordering; broadcasting data tokens in a large parallel system (because of read/write to memory); dispatching instruction tokens in a large parallel system; limitations in data dependency exposure in a program because of memory limitations. These memory limitations are two-fold: one bounded (i.e. synchronizing memory) and the other practically unbounded with VM, i.e. regular memory, which implies a performance hit. In other words, regular memory poses no real limitation but synchroniz-
ing memory does. In conclusion dataflow machines use a bottom-up approach pushing up properties from the architecture to the software level.

2.1.3 UPC

Unified Parallel C (UPC) \cite{28}, designed for high-performance computing (HPC), is aimed at mainstream parallel architectures. The model presents explicit concurrent execution using a single shared, partitioned address space, referred to as Partitioned Global Address Space (PGAS). Each variable may be directly read and written by any computational unit (i.e. processor), but each variable is physically associated with a single processor. UPC is an extension of the C programming language; it implements a front-end extension of the GCC framework with a library invoked during compilation of UPC constructs, in GUPC \cite{29}. UPC constructs map onto hardware processes using high-level threads (Pthreads implementation). The GCC implementation is extended with new language constructs at the front-end and middle-end levels. Even though UPC has a uniform programming model for both shared and distributed memory hardware, it does not capture properly the data dependencies in the SVP model (cf. Chapter \ref{Chapter:3}). The top-down approach of UPC is not to provide a full computing system in opposition to other approaches, but to give a parallel language implementation with an associated compiler implementation.

2.1.4 OpenMP

OpenMP \cite{30,31} is a well-known standard in industry for exploiting parallelism. OpenMP provides directives to the Fortran, C/C++ programming languages to support shared-memory architectures. OpenMP uses code pragmas to expose concurrency in programs, whereas other approaches prioritize language extension with special constructs. A compiler (e.g. GCC with GOMP \cite{32} supports the latest OpenMP 3.0 standards) interprets this concurrency information; OpenMP standards use a dedicated library to support concurrency idioms. With its pragma approach of exposing concurrency in programs, OpenMP pressurizes compilers with concurrency issues at the level of compiler optimizations. The concurrency constructs are not fully exposed to the compiler’s internals which generate gray areas in how to correctly compile concurrent regions. To deal with non-sequential OpenMP code, compiler extensions have been incorporated in the GCC compiler 4.4. Nonetheless, it targets in particular SMPs with coarse-grained concurrency and lacks efficient inter-thread synchronization \cite{33}. Another drawback to use OpenMP is the large variety of code annotations and their options which make the programmability more complex for developers.
2.1.5 Sieve

Sieve [34] can be seen as an alternative to OpenMP, and is released by Coldplay [35]. Sieve is a C++ parallel programming system which targets simplifying the parallelization of concurrent regions. Its approach is with a new construct to encapsulate parallel regions (over loops) to secure memory consistencies. Side-effects within the Sieve block are delayed until the end of the scope (of this encapsulation). As with any other language extension techniques or code annotation techniques, Sieve suffers a little from the fact that an application’s code needs to be redesigned/reengineered. Overheads for developers are not negligible when language constructs are too numerous or complicated to use. Nonetheless, the advantage is the simpleness of this encapsulation. On top of that, the compiler has a clear vision of hazardous areas in the code where compiler optimizations can break code consistencies.

2.1.6 Cilk

Cilk [36] is a general-purpose programming language which is designed for multithreaded parallel computing. The language extends the C programming language with a few language keywords to handle explicit parallelism. In this top-down approach, the programmer identifies concurrent regions which can be run safely. Thread management is performed implicitly by the scheduler. Cilk has a fork-join concurrency approach, but it is not focused on dataflow machines. Moreover, the Cilk run-time scheduler is in charge of mapping concurrency to hardware resources; in contrast to other approaches where constructs map directly to hardware processes or operations. Frigo [37] discusses multithreaded programming in Cilk with a simple Fibonacci example. Investigation on many-core architectures in [38] present the architectural requirements necessary to enable multicore programmability with Cilk. A major advantage of Cilk is a separation of concerns between the parallelism exposure and the resource mapping which permits a Cilk program to run, without rewriting, on any number of processors, including just one.

2.1.7 Google Go

Google Go [39] is an imperative concurrent programming language based on the C programming language and designed by Google. Go defines explicit communication channels between concurrent regions. Go uses a top-down approach which aims at general-purpose parallel machines with shared memory. In terms of implementation, Go uses an extended version of the GCC compiler platform (a new language front-end on top of the compiler and small changes in the middle-end) which is named GCCGo [40]. Concurrent regions are mapped onto goroutines which are executed in parallel with others, including the caller (or creator). A group of goroutines are multiplexed onto multiple threads; execution control is moved between them by blocking them when sending or
Chapter 2. Background in parallel computing systems

receiving messages over channels. Moreover, in current compiler implementation, the goroutines are mapped onto Pthreads which make them heavier than lightweight threads.

2.1.8 Pthreads

The Portable Operating System Interface (POSIX) Threads (Pthreads) API [41] targets shared-memory multiprocessor architectures (e.g. SMPs) and is a set of routines which are incorporated in imperative languages (e.g. C/C++) to expose concurrency. The Pthreads method results in creating high-level threads and requires explicit developer’s attention for thread creation and thread destruction. Workload partitioning and task mapping are explicitly specified by developers, at design-time, in the routine definitions. Moreover, developers must be aware of race conditions and deadlocks in their code when multiple threads access the shared data. To achieve that, the API provides mutexes for mutual exclusion (only one thread enters the critical section) and semaphores (several threads may enter the critical section at the same time). Skillicorn et al. [19] present in further details the existing methods of parallel programming models and languages.

2.1.9 MPI

MPI [42] along with Pthreads [41] probably represent most parallel program methods used for explicit concurrency programming. Kasim et al. [20] made a survey of existing parallel programming models where the pros and cons for each model are decrypted. The MPI approach provides message-passing communication routines among computational processes to model a parallel program running on a distributed memory system. MPI has implementations in several known programming languages, such as C/C++/Fortran, Java, and C#. Similar to the Pthreads method, it is the developer’s responsibility to expose workload partitioning and task mapping. However, ‘thread’ (i.e. process) management is done implicitly whereby it is not necessary to code the creation, scheduling, or destruction of processes. MPI generates then high-level threads where developers have a big responsibility in the way they partition the program’s tasks to achieve good performance.

2.2 Relevant parallel architectures

The ultimate goal of architecture design is the never-ending quest for performance gain. Due to the physical constraints preventing frequency scaling, architecture designers are looking at parallelism to dispatch workloads over multiple computational hardware units. The work in this thesis, the SVP execution model, describes a hardware implementation called the Microgrid in a DRISC.
manner [43]. We have selected several multicore architecture designs that contrast with the SVP execution model introduced in Chapter 3. The UltraSPARC T1 and T2 processors target Internet, database and network servers and are also referred to as Niagara [44] and Niagara II [45]. The latter has 8 cores and supports only 64 threads in hardware. Each core has its own L1 cache, FPU and two ALUs and all cores share a 4MB L2 cache. The major difference between this and a Microgrid is twofold: one is parametric and one fundamental. As described above, a DRISC core can support 100s of cores and 10s of thousands of threads. In programming, Niagara utilizes speculative threading, whereas DRISC manages user or compiler generated concurrency explicitly, with conservative (dataflow) execution rather than speculative execution.

Although the Niagara may appear to be the closest work to that described in Chapter 3, WaveScalar from the University of Washington [27] with its dataflow execution model, is conceptually closer. Both models capture a program’s dependencies and the difference is in the execution model. WaveScalar uses the dataflow firing rule, whereas DRISC uses RISC instruction execution with blocking on register reads. Both models contextualize their synchronizing memory, e.g. via waves in the WaveScalar or in DRISC as concurrent thread contexts in an extended register file. In WaveScalar, all synchronizations are producer-consumer but in DRISC, there is a bulk synchronization on the termination of a family that extends synchronization to the model’s shared memory. Both models define locality statically and create and distribute concurrency dynamically, which are necessary features for efficient compilation and execution. Both models also tolerate high levels of latency and can exploit the latency in a distributed cache-coherent shared memory. This property can greatly reduce the pressure on the processor-memory interface. The conceptual similarities were strengthened recently. In prior work on WaveScalar, the sequential model of programming was managed by providing an ordering on loads and stores but this proved to be too sequential and the model has now been extended with the concept of threads to manage memory concurrency [27].

The other work that is conceptually related to Microthreading (i.e. SVP execution model) is the Data-Driven Multithreading Chip Multiprocessor (DDM-CMP) [46] proposed by the University of Cyprus. DDM-CMP is a coarse-grained dataflow model based on managing blocking threads from the cache interface of a regular processor. Each core in this design has its own L1 cache incorporating a Thread Synchronization Unit (TSU), which implements synchronization on the blocking threads before executing those threads as a conventional code segment. A system network connects the cores and a TSU network connects TSUs. The TSU stores a dataflow graph in its Graph Memory and a number of thread queues, which maintain the status of threads and code blocks. The scheduling of complete threads is done at run-time using the dataflow firing rule and a thread is never issued before all of its operands are ready, at which point it can execute to completion because of the blocking semantics.

The Intel 48-core Single-chip Cloud Computer (SCC) [47] resembles a scalable cluster of computational cores. Intel Labs mentions the incorporation of
technologies intended to scale multicore processors to 100 cores and beyond, such as an on-chip network, advanced power management technologies and support for ‘message-passing’. The Computer Systems Architecture group, at the University of Amsterdam, has received, at the end of 2010 an Intel SCC platform to perform programmability experiments using SVP as execution model.

## 2.3 Modeling concurrency in compilers

Concurrent execution models require an adapted toolchain to render multicore programmability feasible. The way concurrency is exposed in the concurrent source language is a first constraint on how the toolchain is designed. As mentioned in Section 2.1 some concurrent programming languages embed in their design new language construct which explicitly expose the parallelism in an application’s code. Compilers may visualize concurrent regions depending on the exposure of parallelism in a library or with native integration into compiler’s internals. Boehm [48] shows that a thread-based model implemented (Pthreads model is used in the experiments) as a library presents limitations in the use of compiler optimizations. Consequently, these optimizations modify the semantics of the code generated for an application, which may cause them to be unusable. Therefore, problems do not come only with the exposure of concurrency, but also from the way tools are designed. OpenMP [30] uses a pragma approach which is implemented in a compiler with library support. This limits accurate diagnostics and error handling. Adisson et al. [49] mention overheads caused by program design and the sequences of library calls to support concurrent regions. Modeling concurrency in compilers become a problem that cannot be ignored anymore. Gray areas in a program’s code must be reduced to avoid problems; this is a well-known motivation for concurrent programming language design. However, the stress must be also put onto their compiler implementation where gray areas in compiler internal representations must be avoided as well.

Edward Lee, in paper [50], warns of the danger of exposing a thread-based programming model to the vagaries of non-deterministic scheduling in a multiprocessor environment. His own carefully managed software developments revealed bugs that had lain dormant in the code for years until exposed to a multicore environment, where scheduling is no longer a deterministic interleaving. Adapting applications to a new concurrent programming language requires considerable changes to structure or restructure the code of these applications. Gabb et al. [18] discuss new ways of engineering and designing concurrent applications where programming models and their tools are important pieces of the programmability puzzle. Berkeley University, in [8], summarizes the key concepts of the multicore programming menace. The emphasis is also placed on the difficulty to make compilers converge onto multicore technology.

A lot of compilers are available for research (retargeting or reuse). Nonetheless, there are hardly any which consider concurrency assumptions in their
Chapter 2. Background in parallel computing systems

infrastructure. Small-sized compilers such as LCC [51], ACK [52], Tiny C compiler [53] are not reconfigurable frameworks and offer low flexibility to retarget to new architectures and to add new components. Their small size gives the wrong idea of the simpleness for concurrency extensions. Larger compiler frameworks embed a large set of target architectures and a number of aggressive inter- and intra-procedural optimizations. Typically, GCC [54], LLVM [55] offer a good platform for concurrent support with the presence of extension ports. Attempts to use these frameworks for parallel programming have already been tried [56, 57]. However, the very large size of these compilers and their complexity require a considerable amount of time and manpower to achieve production quality. The flexibility and modularity of the compiler infrastructure should offer ways to model and embed concurrency assumptions. This thesis work looks at this question of modeling native concurrency in a compiler infrastructure.

2.4 Requirements for a concurrent execution model

Based on the previous sections, the point in this section is to abstract the essence of previous work in parallel computing in order to conceive requirements for a concurrent execution model. The ultimate goal is the gain of performance for parallel architectures, and consequently to find other ways to save mainstream computing from physical restrictions (the processor and memory walls). Given this, the move to multicore would provide a solution for future computing systems, but the programmability is still a challenge [16, 14]. Backward compatibility of existing applications is of high importance; industry does not want to spend money in redesigning and reengineering applications that are already developed and certified to work perfectly without bugs. The free lunch is over, says Sutter in [13]; the concurrency revolution is here with the presence of multicore architectures [17]. New tools and new thinking from the software industry are necessary to cope with the programmability of these multicore architectures. The concurrency revolution is at all levels of the chain from the architecture up to the programming language including the development tools [8]. Developers want to have a rather easy way of developing for novel parallel computing systems to avoid the nightmares of concurrency programming and design. The main issue is then the need of recompilation for the same application as soon as the target platform changes. The management of concurrency would be more appreciated if it could be opaque to developers and left to the programming paradigm with only the exposure of concurrency. Consequently, the target platform must be able to perform the scheduling and resource management efficiently according to the program’s description.

Moreover, reducing the impact on current computing systems with respect to the introduction of this concurrency would be appreciated. This would render possible the potential reuse of existing proven technology. Therefore, the design of new tools would benefit from former or antecedent technology if this
can be reused in a concurrent context. The main problem remains in the programmability of multicore architectures. So far, sporadic computing systems implement concurrency as a whole paradigm versus uses of code labeling for tagging concurrent areas in a program. The CSA group’s proposition is to visualize a parallel computing system where concurrency and its concerns are treated at different layers of the system.

We consider necessary the following concepts in an execution model for dealing with parallel architectures and parallel programming models. In addition, there is a need for a separation of concerns between resource availability and their mapping to ease off the pressure on developers. Finally, parallel code should scale automatically on different architecture configurations without recompilation. We need a concurrent execution model that can offer and facilitate the following concepts:

- scalability over a distributed multicore architecture,
- programmability of multicore architectures,
- binary code compatibility across arbitrary number of cores (no need to tailor the code across recompilations),
- support concurrency and resource management dynamically.

The following chapter presents the Self-Adaptive Virtual Processor concurrent execution model, called SVP, and its software and hardware implementations - research undertaken by the CSA group. This thread-based execution model captures the previously enunciated concepts and is the basis of the parallel computing system used in this thesis.