The Importance of Synchronization Structure in Parallel Program Optimization

Arjan J.C. van Gemund
Department of Electrical Engineering
Delft University of Technology
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
<http://dutepp0.et.tudelft.nl/~gemund>

Abstract

In automatic, retargetable compilation low-cost, analytic cost estimation techniques are crucial in order to efficiently steer the optimization process. Programming models aimed at optimum expressiveness of parallelism, however, are not amenable to static cost estimation. We present a new co-ordination model, called SPC, that imposes specific restrictions in the synchronization structures that can be programmed. Imposing these restrictions enables the efficient computation of reliable cost estimations paving the way for automatic optimization. Regarding SPC's limited expressiveness we present a conjecture stating that the loss of parallelism when programming in SPC is typically limited to a constant factor of 2 compared to the unrestricted case. This limited loss is outweighed by the unlocked potential of automatic performance optimization as well as the portability that is achieved. We demonstrate how SPC enables automatic program optimizations through a compilation case study involving a line relaxation kernel on a distributed-memory machine.

1 Introduction

A long-term goal in parallel program compilation is the development of completely *automatic* program optimization techniques without any user interaction, thus providing portability as well as performance. Based on an explicitly parallel algorithm description the compiler should be able to automatically determine, for example, the usefulness of computation/communication pipelining, the appropriate loop transformations to optimize the use of memory hierarchy, or an acceptable data layout without having the user supply data distribution directives, all of which would require a major understanding on the programmers part of the complex interplay between parallel program and machine.

In [27] an optimization framework is described based on the use of an expert systems approach. The various optimization rules fire as a result of the availability of certain machine features (e.g., the existence of memory hierarchy, loop self-scheduling) depending on the actual search context.

A different approach, in which the optimization problem is expressed (and solved) in terms of a global execution time cost minimization problem, is described in [20] where the optimum data layout of multi-phase algorithms on distributed-memory machines is computed.

From a compiler engineering point of view, using cost minimization techniques provides a relatively clean way to separate the optimization technique itself from a description of a machine's peculiarities. The latter characteristics are conveniently abstracted in terms of a generic cost modeling technique that accounts for the performance effects of program modifications. For instance, if a large data locality results from a certain loop interchange its favorable effects for a machine that happens to have caches, will automatically show up in the cost model associated with the transformed program version. The optimization technique itself, however, is machine-independent. The trade-off is cost estimation at compile-time. In that respect, a rule-based approach can be considered a short-cut in which the effect of a machine feature on an optimization decision is hard-wired rather than computed at compile-time. Consequently in cost-driven optimization the cost estimation effort itself is a crucial success factor.

As many of the specific optimization decisions interfere with one another in terms of the overall performance result, cost models must have a global scope. Therefore, the key factors of the underlying cost model are not only its *accuracy* but also its solution *complexity*, especially in view of the huge number of cost evaluations that are possibly made in the course of the optimization process. This immediately narrows the choice of cost estimation techniques to analytic techniques that yield explicit (closed form) models, thereby ruling out more accurate analytical techniques based on stochastic models such as queuing nets, Petri nets, and process algebras. Currently a range of fine static cost prediction methods exists [3, 7, 10, 28]. However, either the underlying analysis and associated parameter space is targeted to a particular type of parallel architecture, or the technique is not designed to produce reliable estimates in extreme points in the entire parameter search space. This might lead to wrong optimization decisions as erratic parameter solutions may well be temporarily generated in the course of an automatic optimization process.

The heart of the problem with respect to the existence of closed form, yet reliable cost modeling techniques at compile-time is the choice of parallel programming model. By tradition, parallel programming models are focused on *expressiveness*, i.e., the ability to express the inherent parallelism within the algorithm to the ultimate, rather than *perform-*

ance *analyzability*, which in this context means the ability to predict the actual parallel computation’s execution time in terms of closed-form, analytic cost models with optimum cost/performance. We feel that the current trend in the development of parallel programming models may not sufficiently reflect the importance of performance *analyzability* which is the crucial requirement for fully automatic, retargetable compilation.

Aimed towards the development of an automatic program optimization framework, this paper presents the following contributions:

- We describe a new coordination model, called SPC. By imposing specific *restrictions* in the synchronizations that are allowed, a performance *analyzability* is achieved that enables efficient optimization.
- We show how the SPC model captures the essential parallelism in parallel algorithms, and we present a conjecture that the loss of parallelism due to the above restrictions is typically limited to a factor of 2 compared to the unrestricted case.
- We demonstrate how SPC’s associated cost estimation technique enables automatic program optimization through a case study in which we derive a well-known optimization for a well-known line relaxation algorithm (ADI) on a distributed-memory machine.

The “SP” prefix in SPC applies to the fact that the algorithm must be described in terms of a *series-parallel* (SP) structured computation, which implies structure with respect to the *condition synchronization* [2] (CS) patterns that are possible (only SP DAGs). The “C” term refers to the structured use of *mutual exclusion* [2] (ME) in *contention* for processing resources, which, combined with a material-oriented [21] programming paradigm, we coin *contention programming*¹. Although the concept of material-oriented simulation as well as the use of structured operators for CS (e.g., fork/join programming) and ME (e.g., monitors) are well-known, to the best of our knowledge the *combination* of these concepts to create a highly structured parallel programming model has not been introduced before.

While the question whether SPC delivers sufficient expressiveness is clearly at the heart of our approach, recent results supporting our conjecture are presented elsewhere [9]. In this paper we will primarily concentrate on the performance *analyzability* aspect of the SPC approach. In Section 2 we present the SPC programming model as well as an elaborate rationale underlying our specific trade-off between expressiveness and *analyzability*. In Section 3 we show how SPC provides a framework for automatic optimization. In particular, we describe a case study in which we consider a number of optimizations for a line relaxation algorithm on a distributed-memory machine. In Section 4, we review some of the related work in the field of parallel programming models in order to put our approach into perspective. Finally, in Section 5 we summarize our work.

¹The original terminology *material-oriented* modeling, and its dual, *machine-oriented* modeling, stem from the domain of simulation of, e.g., plant production lines. We feel that the application of material-oriented modeling in the distinct domain of parallel programming justifies using the specific name “contention programming”. Unlike its dual, in the contention modeling approach the data computations/flow (i.e., the algorithm) is modeled by active *processes* while the machine is modeled as a passive collection of *resources*, being shared by *contending* processes.

2 The SPC programming model

2.1 Language

SPC is not a full fledged programming language that would very much require the specification of the data *computation*. It specifies a programming paradigm with respect to the *coordination* of the program’s process parallelism. To facilitate the presentation, however, we shall loosely introduce a simple language implementation to provide a description vehicle. For such a language we borrow a number of programming concepts used in PAMELA, a PerformAnce Modeling LAnguage [13], that is well-suited to support the SPC approach, especially as we will only be concentrating on the control flow aspects of the parallel computation, not the data processing itself. Apart from some syntactic sugar, the syntax used for SPC conforms to the PAMELA language implementation [22]. Although PAMELA has indeed been developed with the SPC paradigm in mind, note that the SPC approach represents a distinct model of parallel computation that is independent of the underlying description language. Furthermore, being a full-fledged performance simulation language, PAMELA features much more operators than the few we will use to explain the principles of the SPC programming model.

2.1.1 Processes

Like PAMELA, SPC is based on a *process-algebraic* specification model. This implies that SPC programs are a set of functional process *equations*. Programming, i.e., applying compositions is done by specifying process equations using a straightforward substitution mechanism. In order to ensure the correct binding, `{,}`’s can be used to delimit compound process expressions. Apart from data types needed to express numeric computations, the main data type in SPC is the *process*.

Basic data computations are expressed in terms of processes. For example, the equation

```
update(i) = __C__ { A[i] = A[i-1] + A[i+1]; }
```

describes a process `update(i)` where the `__C__ { ... }` is an inlining facility (as implemented in PAMELA) to provide an interface with actual data computation host languages (C in this case).

An SPC program consists of a set of process equations that through substitution constitutes one parallel process expression. By convention, the expression tree is rooted by a special process called `main`, that represents the overall program.

The *process* abstract data type comes with the following operators:

- **sequential composition**
A sequential composition is described by the infix operator `;` like in `task_1 ; task_2`. Sequential replications (“loops”) are described by the reduction operator `seq` as in `seq (i = 1, N) task(i)` which is equivalent to `task(1) ; task(2) ; ... ; task(N)`. Note that this definition provides a basic form of CS.
- **parallel composition**
A parallel composition is described by the infix operator `||` like in `task_1 || task_2`. Parallel replications (“loops”) are described by the reduction operator `par` as in `par (i = 1, N) task(i)` which is equivalent to `task(1) || task(2) || ... || task(N)`. The parallel operator has a fork/join semantics which implies

a mutual barrier synchronization at the finish of the parallel section for each task involved. Note that this implies a *structured* form of CS. Also note that SPC allows dynamic parallelism as the value of N may be controlled at run-time.

- conditional composition

A conditional process is specified by the `if` operator as in `if (c) task_1 else task_2` where the `else` part is optional.

Finally, SPC also includes a `while` construct, which, however, is not considered in this paper.

It is important to note that since the SPC coordination model is intended as the essential mechanism to explicitly control parallelism, it does not take into account *implicit* forms of synchronization that would be required as a result of data dependencies in the program. Hence, if data dependencies need to be obeyed, it is the explicit responsibility of the programmer to include the appropriate SPC constructs to ensure program correctness. As mentioned earlier, in SPC we focus on the benefits of *explicit* and *structured* synchronization, rather than on compile-time data dependence analysis for programming convenience or integrity.

2.1.2 Resources

A central philosophy within SPC is that parallel computations are expressed in terms of *processes* (the computations) and *resources* (the computation providers), the latter introducing *limitations* with respect to the actual parallelism in the system. While the *process* abstract data type provides a means to specify both parallelism and CS, the *resource* abstract data type provides the means to impose ME. A process that is to be executed under ME is assigned to a resource as in

```
f, g -> some_resource
```

where the `->` operator denotes the mapping of processes `f` and `g` to processor `some_resource`. Here `f` and `g` are executed under ME as a result of the fact that they are mapped onto the same resource. Hence, in the program

```
main = f || g      ! f, g = __C__ { ... }
f, g -> some_resource
```

the execution of `f` and `g` are effectively serialized. Note, that again synchronization is structured. A resource can either be logical, e.g., model a critical SW section, or physical, e.g., model a CPU, memory bank, communication bus, etc. The scheduling policy we will consider in this paper associated with resources is simply FCFS with non-deterministic, fair conflict arbitration but other scheduling disciplines can also be specified [15].

The notion of resources is universal in SPC, i.e., in principle, *each* process in SPC is always mapped onto at least one resource². The underlying concept is that a process must always execute in the context of some resource (an instruction will cost cycles to at least one of more resources). However, the declaration is usually not required as most processes are meant to be mapped to their own exclusive processing resource anyway. Hence, many SPC models will not require the explicit use of the resource limitation mechanism.

In summary, the SPC model of coordination is based on only a few constructs, i.e., the `;`, `||`, and `if` process composition operators and the `->` resource assignment. Note that

²As multiple mappings may be specified a process may simultaneously use a set of resources [15].

within SPC the resource concept is only used to express ME in order to express dynamic synchronization between program level components. Although possible, it is *not* intended to direct the actual mapping of processes onto physical processors or other machine resources as in our aim to study automatic program optimization the program description is meant to be completely machine-independent. The actual implementations involving the mapping onto physical machine resources are *only* introduced in the course of deriving program optimizations, in which SPC plays the role of intermediate language.

2.2 Example

In the following we demonstrate the use of SPC by discussing a pipelining example. An elaborate presentation including examples of how reduction, divide-and-conquer, and dynamic scheduling are expressed in SPC appears in [16].

Because of the limited expressiveness of the data parallel programming model there is currently a great deal of interest into extending this programming model to allow the expression of other forms of parallelism (see Section 4 on related work). Software pipelining is often used as motivating example. The pipelining example shows an essential difference in the SPC contention programming approach compared to current task parallel approaches to pipelining. Unlike usual message-passing solutions, in the SPC model, pipelining is still expressed in terms of original data parallelism (which it in fact is), yet being *constrained* in terms of the available computing *resources*.

Consider a simple data parallel application expressed by the following process equations

```
main =      par (i = 1, N)
            proc(i)
proc(i) = seq (j = 1, M)
            op(i,j)
op(i,j) = __C__ { f(imag[i],j); }
```

which represents a parallel section where each process `proc` executes a sequence of operations `op(i,j)` on some data structure `imag[i]` (e.g., some image computation). Being a process-algebraic language, the program could also have been described by the single expression

```
main = par (i = 1, N)
        seq (j = 1, M)
        op(i,j)
```

where `op()` is defined as before.

In the pipelined scheme we restrict the actual parallelism from N to M . Typically, the solution is programmed through the use of a message-oriented model in which each of the M processing stages is programmed in terms of a producer-consumer scheme. In contrast, our SPC solution is given by

```
main = par (i = 1, N)
        seq (j = 1, M)
        op(i,j)
op(i,j) -> unit(j)
```

In this approach we still specify the inherent data parallelism of N in the algorithm just like before. However, we introduce a *constraint* by mapping each individual function `op(i,j)` to M resources called `unit(j)`, yet without *overspecification*, i.e., without imposing some *static* scheduling order regarding `i` other than the standard dynamic FIFO

constraint associated with the FCFS resource class. The pipeline behavior directly follows from the ME induced by this resource assignment. Note that this is actually a very natural way of programming that is also very portable. In essence, the *main* equation still specifies the intended *algorithm*, that is essentially data parallel. The pipeline *implementation* simply follows from the fact that the operations are mapped at program level to only M virtual processing resources, which, by nature, impose ME with respect to their service. Thus, the synchronization mechanism we use is based on the fact that each data operation $\text{op}(i, j)$ actively “contends” for the available computing resources, hence, the name “contention programming”, a concept that lies at the heart of the SPC programming model.

Note that the SPC solution still involves the use of N processes instead of M where N can be extremely large. As mentioned earlier, however, in our approach we assume the existence of an automatic mapping layer that supports this “process virtualization” in order to provide a truly portable programming model. In [16] it is shown that the compile-time transformation to an efficient M -process SPMD implementation is completely straightforward and can be expressed in terms of SPC itself which illustrates that SPC is also useful as an intermediate, process-algebraic transformation formalism.

2.3 Cost Estimation

In this section we briefly describe the analytic cost estimation process associated with SPC programs that is based on the use of PAMELA. As mentioned earlier, PAMELA is almost equivalent to the above SPC language, be it that it is a *performance* modeling formalism. As such it incorporates the notion of virtual time. A detailed description of the PAMELA language appears elsewhere [13, 15].

Because of the large similarity the mapping of SPC expressions to PAMELA expressions is straightforward. The time cost of an SPC expression is determined by mapping the corresponding PAMELA expression to a closed-form, *symbolic* expression in the time domain. This automatic transformation process is *linear* in the *symbolic* size of the program. It is based on a critical path analysis combined with an approximation of the serialization that may occur due to ME and is described elsewhere [14, 15].

For instance, the SPC pipeline example

```
main = par (i = 1, N)
      seq (j = 1, M)
        op(i, j)
op(i, j) -> unit(j)
```

is automatically mapped to the symbolic time cost estimate

$$T = (N + M - 1)\tau_u$$

where τ_u denotes the time cost of operation $\text{op}(i, j)$ on $\text{unit}(j)$ [15]. Although the above cost expression happens to be correct, the estimation process produces a lower bound. Yet, unlike cost estimation approaches based on analyzing CS only, all the appropriate order terms are essentially generated in the cost function. In [15] it is shown that on average the cost estimation error due to the effects of synchronization is typically limited within tens of percents, regardless the size and shape of the model. Thus PAMELA’s cost/performance properties form a convenient basis for automatic optimization through comparative cost modeling.

2.4 Rationale

As mentioned earlier, the choice of programming model involves a trade-off between expressiveness and performance analyzability. This is shown in Fig. 1 that characterizes the process of mapping the original problem to an explicitly parallel algorithm, which, in turn is mapped to the target parallel machine. Depending on the programming model, in the first mapping stage (“programming”) a loss of parallelism may occur such that the critical path T inherent in the problem is increased to T/α (α denotes the fraction of inherent problem parallelism that is captured by the program). Similarly, the choice of programming model determines the “degree” (γ) in which symbolic cost estimations (SCE in the figure) can be generated from the program, that, in turn, entails an upper bound on the quality of the automatic optimization that can be applied during the compilation stage (β in the figure which represents the fraction of actually utilizable machine parallelism). In the following sections we present the rationale for the SPC programming model both from the perspective of performance analyzability (γ) as well as expressiveness (α).

Figure 1: Program as intermediate between problem and implementation

2.4.1 Analyzability

Due to the use of a critical path analysis technique, the SP synchronization structure of the SPC computation is essential in order to enable symbolic estimates that are at least closed form expressions. However, the form in which scheduling non-determinism is expressed is also crucial with respect to (static) analyzability. In order to appreciate the significance of this form, let us consider a message-oriented model such as CSP [19] where scheduling non-determinism is achieved through the choice operator ‘+’ (or ‘□’). In [16] a comparison is given between the SPC solution and a CSP-type solution for a simple client-server problem. Compared to the SPC solution, the CSP-type solution is much less amenable to a mechanized analysis scheme that produces a symbolic cost estimate. Especially when message-oriented programs become complex, it is impossible to efficiently deduce the dominant “thread of synchronization” that may run across multiple processes, partly routed through the non-deterministic + operator. This fundamental problem is due to the use of the *low-level* + operator, in combination with providing *separate* constructs for sending and receiving. In SPC, CS and ME are kept orthogonal, both in terms of *single*, high-level operations. Thus the need to first “reverse engineer” to a higher level model, which is impossible to mechanize, is completely avoided. Another striking example of the problems associated with analyzing message-passing programs is presented in [15].

2.4.2 Expressiveness

Due to the choice of high-level synchronization constructs, expressiveness will be less than, e.g., a message-oriented formalism. In order for SPC to be an adequate alternative model of computation, clearly, the loss of parallelism (α) must be small enough. For example, consider the NSP (non-SP) computation shown on the left of Fig. 2. One of the pos-

Figure 2: Possible SPC version of original NSP graph

sible SPC versions of this NSP computation, in which none of the original precedence relations are violated, is shown on the right. One can easily verify that the critical path of the SPC version has increased. In general, the crucial question becomes how much inherent parallelism we sacrifice *at maximum* by imposing the SPC model. The essential rationale for SPC with regard to the expressiveness of parallelism is loosely stated in terms of the following conjecture.

Conjecture 2.1 *Let G be a parallel computation. Let T_G denote the minimum critical path of G when expressed in a programming model that does not impose restrictions with respect to the synchronization constraints inherent in the problem. Let G' be an SPC program that also computes G but has a critical path $T_{G'}$. Then we conjecture that for non-pathological workload distributions it holds*

$$\forall G \exists G' : \frac{T_{G'}}{T_G} \leq 2$$

Argument: The argument breaks down into the following parts (see [16, 9] for more details):

- **Equivalence.**

Many problems are naturally stated in terms of parallel algorithms composed of possibly nested parallel sections and reductions (many data parallel operations, divide-and-conquer, client-server systems, pipelines). As shown in [16] this class of problems can be expressed in SPC, i.e., without any loss of parallelism. It is also shown that the scheduling non-determinism needed in order to describe the dynamic scheduling needed in order to avoid *a priori* loss of parallelism, is adequately provided by SPC's ME construct.

- **Approximation.**

Although many computations can be expressed in SPC *without* any loss of parallelism, many computations remain that are justifiably NSP structured in order to achieve maximum parallelism. For this class of NSP computations we conjecture that $T_{G'} \leq 2T_G$. As a simple example, consider the NSP graph in Fig. 2. One can easily verify that the critical path of the SPC version is at most twice the length of the original critical path, *regardless* the actual work loads of all tasks. The

extreme situation only occurs in the pathological case where task 1 and 4 have equal, finite work load of τ units time, while the other tasks have zero work load. While $T_G = \tau$ the SPC version yields $T_{G'} = 2\tau$ due to the loss of parallelism. Although it may seem that the bound depends on the number of nodes involved it is ultimately determined by topology. In [9] it is shown that many NSP DAG topologies such as the parallelism/reduction graphs occurring in LU factorization can be converted to SPC form well within a factor 2 loss of potential parallelism, while for certain DAG topologies the factor 2 bound is only exceeded under extremely unlikely (pathological) task workload distributions.

□

The above conjecture has been a major inspiration for the SPC approach. In order to substantiate this conjecture, research aimed at deriving proofs as well as supporting empirical evidence has recently started. Preliminary empirical data, based on the results of a polynomial-time algorithm [9] converting and comparing NSP graphs to SP versions indeed provide compelling evidence in support of the above conjecture.

It is important to note that the factor 2 upper bound may seem as a drastic performance reduction when compared to alternative programming models with an expressiveness comparable to, e.g., CSP. We believe, however, that this sacrifice is justified by the potential of fully automatic, cost-driven optimization. Even when the initial loss of parallelism is not entirely compensated for by the increased optimization abilities, the essential portability will outweigh the effort spent on realizing T_G every time through manually porting the code to the latest machine architecture.

3 Compiling Optimizations

In this section we demonstrate how SPC is used for the compilation of optimization tests. First, we introduce the technique by discussing vectorization. Next, we look at a more realistic example involving an ADI line relaxation algorithm on a distributed-memory machine. In the examples we will use SPC as an intermediate formalism in terms of which the optimizations will be described. The associated performance models are derived using the PAMELA cost calculus.

3.1 Vectorization

Consider the following data parallel computation

```
par (i = 1, N)
  mult(i)
```

where the `mult` task represents some scalar multiplication (ignoring data traffic). Depending on the target machine resources available, there may be various implementations possible. As the above SPC description is independent of the underlying architecture, we can consider these mappings simply in terms of defining the implementation of `mult` depending on the instructions available on the (virtual) machine.

Scalar processing simply implies the following mapping

```
mult(i) -> alu
```

Thus all N `mult` operations are implemented by the same computational unit `alu`. Due to the sequentialization as a result of the ME constraint this directly generates the cost estimate

$$T_s = N\tau_s$$

where τ_s denotes the scalar processing time of the unit. The s subscript in T denotes the scalar implementation.

Alternatively, vector processing implies the existence of a vector processing resource that corresponds to the following implementation

```
mult(i) = seq (j = 1, M)
           sub_mult(i,j)
sub_mult(i,j) -> unit(j)
```

as discussed in the pipelining example. The PAMELA calculus directly generates the cost estimate

$$T_v = (M + N - 1)\tau_c$$

where τ_c denotes the effective cycle time such that the combination of M and τ_c account for both startup time and bandwidth as measured externally, including software overhead and memory latency. The v subscript in T denotes the vectorization alternative.

Given the above alternatives, the optimization test compiles to $T_v < T_s$ which can be symbolically reduced at compile-time according to

$$T_v < T_s = \tau_s + N\tau_c < N\tau_f = N \geq \left\lceil \frac{\tau_s}{\tau_f - \tau_c} \right\rceil = N \geq N^*$$

where N^* is the cross-over value usually hard-wired in compilers.

The example shows two aspects. First, it demonstrates the *abstract* SPC approach towards the choice of mapping the (inherently) parallel algorithm onto the scalar or vector machine resource, merely by discussing alternative `mult` implementation models while using the *same* algorithmic description. Note that an explicitly *sequential* implementation of the algorithm in the scalar case has *not* been specified. It simply followed from the contention mechanism. Second, it shows how the optimization problem can be expressed in terms of low-complexity cost models. Even though the generic optimization scheme is based on the use of unreduced cost estimation expressions, due to their algebraic nature mechanical reduction is possible yielding low-cost optimization tests. Even when N is only known symbolically, the optimization test is evaluated at run-time at negligible expense. The decision then to make at compile-time is merely the question if it is worth-while to compile the additional integer test $N \geq N^*$.

For brevity, the above discussion did not include the performance effects of the memory system (e.g., memory pipelines based on parallel memory bank access). However, we stress the fact that this in no way complicates the optimization scheme other than adding some syntactic complexity.

3.2 Case Study

In this section we will demonstrate the SPC approach applied to the well-known, data parallel line relaxation algorithm kernel (ADI) for a distributed-memory machine [1]. First, we show how the optimal data partitioning is derived considering the possibility of remapping. Next, we show how the optimal implementation is derived, based on an inter-processor pipelining scheme, instead of remapping between

the two phases of the algorithm. This solution is similar to the compiler optimization discussed in, e.g., [1]. Yet, we are able to automatically derive this optimization within the SPC framework.

3.2.1 Data Partitioning

Consider both phases of the line relaxation applied to an $N \times N$ matrix A , expressed in SPC according to

```
seq (i = 1, N-2)           ! sweep vert.
  par (j = 0, N-1)
    update_v(i,j)
;
seq (j = 1, N-2)           ! sweep horiz.
  par (i = 0, N-1)
    update_h(i,j)
```

where `update_v(i,j)` computes $a_{i,j} = f(a_{i-1,j}, a_{i+1,j})$ and `update_h(i,j)` computes $a_{i,j} = f(a_{i,j-1}, a_{i,j+1})$. First we will only consider the data partitioning for the vertical phase. Later on, we will consider both phases.

In the vertical phase the relaxation sweep direction is in the i direction. In the parallelization for a P processor distributed-memory machine we consider the choice between two regular block partitioning strategies, i.e., either along the i axis or along the j axis (a choice that is trivial from a human point of view).

Of course, one could consider the corresponding SPMD message-passing implementation as a modeling basis in order to determine which alternative has the lowest execution cost. However, the choice for using implementation-level cost modeling as compiler feedback has great disadvantages. While for the j axis partitioning the syntax of an SPMD implementation would clearly reveal the potential speedup, for the i axis partitioning the generated message-passing code does *not* clearly reveal the fact that the synchronizations of the generated message-passing scheme will still *sequentialize* the entire computation (see [15] for details). Again, this illustrates the problems associated with the choice of a message-passing scheme as the basis for analytic performance optimization as described in Section 2.4.1.

The SPC description, in contrast, directly reveals the correct optimization. Let the mapping function $\mu(i, j)$ denote the index of the processor resource responsible for the update of a_{ij} . In terms of the above SPC model this implies the mapping

```
update_v(i,j) -> cpu(mu(i,j))
```

Let τ_u represents the computation work load associated with the update of element a_{ij} on node $\mu(i, j)$. Let $B = N/P$ denote the block size where for simplicity we assume $P|N$.

For the j axis partitioning it holds $\mu(i, j) = j/B$. From the PAMELA analysis it directly follows

$$T = (N - 2) \frac{N}{P} \tau_u$$

which implies speedup. For the i axis partitioning it holds $\mu(i, j) = i/B$. It follows

$$T = (N - 2) N \tau_u$$

which, unlike the message-passing SPMD code, directly reveals the algorithm's sequential nature. Indeed, in the present algorithm setting, an i axis partitioning will not yield any speedup.

3.2.2 Pipelining

In [16] symbolic cost estimations are derived that determine whether not it is advantageous to apply remapping between both phases of the algorithm. This derivation involves the use of communication models. Due to space limitations, however, in the paper we only consider an alternative optimization that is usually much more attractive than remapping. This optimization is based on pipelining the computation across the processors during the horizontal phase which is explained at length in [1]. Unlike in the remapping case, we ignore data communication for simplicity.

Recall the original code for the horizontal phase, i.e.,

```
seq (j = 1, N-2)
  par (i = 0, N-1)
    update_h(i, j)
```

For the purpose of a future algorithm transformation we will consider the following equivalent version

```
seq (p = 0, P-1)
  par (i = 0, N-1)
    update_b(i, p)
```

where the j loop is aggregated to P blocks of size B , which, under the original data mapping, are precisely local to processor p according to

```
update_b(i, p) = seq (j = 0, B-1)
                  update_h(i, p*B+j)
update_b(i, p) -> cpu(p)
```

The PAMELA analysis directly yields

$$T = (N - 2)N\tau_u$$

which, as before, predicts the absence of speedup.

We will now consider a transformation based on a simple *loop exchange* according to

```
par (i = 0, N-1)
  seq (p = 0, P-1)
    update_b(i, p)
```

that is allowed as the i and j references are independent. Note that this immediately produces a *pipeline* similar to the earlier examples. Indeed, instead of running sequential, each i loop is now pipelined such that the next processor executes a different i loop instance concurrently, yet obeying the j sequence [1]. The next i loop is executed when the previous j loop has traversed exactly one processor, i.e., a block of B indices. The reason for the previously introduced block aggregation of the j loop is that simply exchanging the original loops would formally not produce the intended schedule. Because of the fair contention mechanism the (i, j) accesses would still be scheduled according to a column-major scheme rather than the intended row-major scheme.

The loop exchange has a great impact on the cost estimate as from the PAMELA analysis it directly follows

$$T = (N + P - 1)B\tau_u = \mathcal{O}\left(\frac{N^2}{P}\right)\tau_u$$

Thus, simply by exchanging loops in the SPC model, the same order of performance can be achieved as in the first phase, *without* remapping the data. Clearly, the comparison between remapping ($\mathcal{O}(N^2)\tau_u$) and pipelining ($\mathcal{O}(N^2/P)\tau_u$) would be reduced at compile-time in favor of pipelining. Of course, the additional communication overhead during the

pipelining is ignored. Especially when many sweeps are performed during both phases remapping can still be appropriate. This issue is elaborated in [15] which includes the communication effects into the analysis.

The above case study illustrates how the various optimization possibilities are tested by automatically generating the cost models and comparing and reducing them to simple tests. Due to the SPC framework this process is algebraic and simple that permits this generic compile-time/run-time approach towards cost-driven program optimization.

4 Related Work

In the following we briefly discuss some of the interesting parallel programming models that have been proposed, and review them in terms of the trade-off they represent with regard to expressiveness and performance analyzability. For an extensive review on the subject of static (and dynamic) cost estimation one is referred to [15].

As mentioned in the introduction, many programming models (or coordination models) do not impose any restrictions with regard to expressiveness, such as the message-oriented programming models (e.g., Strand [12], CSP [19], MPI [24], PVM [25]). As mentioned in the rationale in Section 2, the problems due to NSP synchronization structure as well as the low-level at which scheduling non-determinism is provided, effectively rule out symbolic cost analysis. On the other hand, synchronization restrictions such as, e.g., in Fortran-M in order to guarantee determinism [11] may lead to scheduling inflexibility. This also arises in functional data parallel models (e.g., SCL [8]) that provide deterministic coordination skeletons.

In more procedure-oriented programming models synchronization is achieved through the use of shared objects like the queues in Orca [18], the sync objects and atomic functions in CC++ [5], the monitor concept in Opus [6], and the tuples in Linda [4]). With their semaphore-like full-empty semantics, queues and tuples provide a means to express both CS and ME without imposing any synchronization structure. The CC++ and Opus constructs for atomicity are more or less comparable to our structured approach to provide a higher-level form of scheduling non-determinism through ME. The flexibility offered by CC++ and Opus with respect to CS, however, does allow for the specification of NSP graphs which complicates symbolic cost analysis. Other approaches that also have the ability to produce NSP structure are models where task parallelism (i.e., CS) is implicitly deduced from the data dependencies. Languages according to this model include Fx [17] and Jade [23]). Again, the primary aim of these approaches is at capturing maximum (data flow) parallelism, rather than imposing *a priori* synchronization restrictions to facilitate performance analyzability.

5 Conclusion

We have presented the SPC parallel programming model and associated cost calculus that is inspired by the wish to balance expressiveness and performance analyzability to provide a framework for automatic optimization. This balance is achieved by imposing restrictions regarding synchronization structure in conjunction with a programming concept, called contention programming. Although this concept as well as the use of structured operators for CS (SP programming) and ME (monitors) is not new, to the best of our knowledge the combination of these concepts to create a

highly structured parallel programming model has not been introduced before. As a crucial part of the rationale for SPC we have presented a conjecture that the loss of parallelism due to the SPC synchronization constraints is bounded by a factor of 2 while the potential of truly automatic optimizability comes within reach. We have demonstrated the advantages of SPC by deriving a number of optimizations for a line relaxation algorithm on a distributed-memory machine. Future work will be directed towards an implementation of the automatic optimization framework underlying SPC, to establish the specific advantages SPC offers with regard to automatic partitioning and scheduling, as well as to verify our above conjecture for various problem classes. Parts of the SPC philosophy will be applied within the AUTOMAP research project [26] that aims at a parallel programming language and compilation system in which code and data mapping are completely automated.

Acknowledgements

I am most grateful for the valuable feedback I received from the referees.

References

- [1] V.S. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, J. Mellor-Crummey, S. Warren and C-W. Tseng, "Requirements for data-parallel programming environments," *IEEE Parallel and Distributed Techn.*, Fall 1994, pp. 48–58.
- [2] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no. 24, 1983, pp. 132–145.
- [3] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *Proc. 3rd ACM SIGPLAN Symposium on PPOPP*, Apr. 1991.
- [4] N. Carriero and D. Gelernter, "Linda in context," *CACM*, vol. 32, Apr. 1989, pp. 444–458.
- [5] K.M. Chandy and C. Kesselman, "CC++: A declarative concurrent object-oriented programming notation," in *Research Directions in Concurrent Object-Oriented Programming* (G. Agha, P. Wegner and A. Yonezawa, eds.), MIT Press, 1993, pp. 281–313.
- [6] B. Chapman, P. Mehrotra, J. Van Rosendale and H. Zima, "A software architecture for multidisciplinary applications: Integrating task and data parallelism," in *Proc. Fifth Workshop on Compilers for Parallel Computers*, Malaga, June 1995, pp. 454–466.
- [7] M.J. Clement and M.J. Quinn, "Multivariate statistical techniques for parallel performance prediction," in *Proc. 28th Hawaii Int. Conf. on System Sciences, Vol. II*, IEEE, Jan. 1995, pp. 446–455.
- [8] J. Darlington, Y. Guo, H.W. To and J. Yang, "Functional skeletons for parallel coordination," in *Proc. Europar'95 (LNCS)*, 1995, pp. 55–69.
- [9] A. González Escribano, Valentín Cardenoso Payo and A.J.C. van Gemund, "On the loss of parallelism by imposing synchronization structure," in *Proc. 1st EURO-PDS Int'l Conf. on Parallel and Distributed Systems*, Barcelona, June 1997.
- [10] T. Fahringer and H.P. Zima, "A static parameter-based performance prediction tool for parallel programs," in *Proc. 7th ACM ICS*, Tokyo, July 1993, pp. 207–219.
- [11] I. Foster, "Task parallelism and high-performance languages," *IEEE Parallel and Distributed Technology*, Fall 1994, pp. 27–36.
- [12] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.
- [13] A.J.C. van Gemund, "Performance prediction of parallel processing systems: The PAMELA methodology," in *Proc. 7th ACM ICS*, Tokyo, July 1993, pp. 318–327.
- [14] A.J.C. van Gemund, "Compiling performance models from parallel programs," in *Proc. 8th ACM ICS*, Manchester, 1994, pp. 303–312.
- [15] A.J.C. van Gemund, *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, Apr. 1996.
- [16] A.J.C. van Gemund, "Notes on SPC: A parallel programming model," Tech. Rep. 1-68340-44(1997)03, Delft University of Technology, Apr. 1997.
- [17] T. Gross, D.R. O'Hallaron and J. Subhlok, "Task parallelism in a high-performance Fortran framework," *IEEE Parallel and Distributed Techn.*, Fall 1994, pp. 16–26.
- [18] S. Ben Hassen and H. Bal, "Integrating task and data parallelism using shared objects," in *Proc. 10th ACM ICS*, Philadelphia, July 1996, pp. 317–324.
- [19] C.A.R. Hoare, "Communicating Sequential Processes," *CACM*, vol. 21, Aug. 1978, pp. 666–677.
- [20] Ulrich Kremer, "NP-completeness of dynamic remapping," in *Proc. 4th Int. Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993, pp. 135–141.
- [21] W. Kreutzer, *System simulation, programming styles and languages*. Addison-Wesley, 1986.
- [22] M. Nijweide, "The PAMELA compiler," Tech. Rep. 1-68340-28(1996)08, Delft University of Technology, Delft, The Netherlands, Sept. 1996.
- [23] M.C. Rinard, D.J. Scales and M.S. Lam, "Jade: A high-level, machine-independent language for parallel programming," *Computer*, June 1993, pp. 28–38.
- [24] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*. MIT Press, 1996.
- [25] V. Sundaram, "PVM: A framework for parallel distributed computing," *Concurrency-Practice and Experience*, vol. 2, Dec. 1990, pp. 315–339.
- [26] K. van Reeuwijk, H.J. Sips, H.X. Lin and A.J.C. van Gemund, "Automap: A parallel coordination-based programming system," Tech. Rep. 1-68340-44(1997)04, Delft University of Technology, Apr. 1997.
- [27] K-Y. Wang, "Intelligent program optimization and parallelization for parallel computers," Tech. Rep. CSD-TR 91-030, Purdue University, Apr. 1991.
- [28] K-Y. Wang, "Precise compile-time performance prediction for superscalar-based computers," in *Proc. ACM SIGPLAN PLDI'94*, Orlando, June 1994, pp. 73–84.