UNIVERSITY OF TWENTE.

A VERIFICATION TECHNIQUE FOR DETERMINISTIC PARALLEL PROGRAMS

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

JOINT WORK WITH SAEED DARABI AND STEFAN BLOM

THE CHALLENGE OF RELIABLE SOFTWARE

ICT problems Dutch gouvernment

Toyata Prius: software errors due to lack of testing

Unreachable banks because of network problems

Mars Climate Orbiter: Crash due to different units

SPECIFYING PROGRAM BEHAVIOUR

Use logic to describe behaviour of program components

- Precondition: what do you know in advance?

  Example: increaseBy(int n)

    requires n > 0

- Postcondition: what holds afterwards

  Example: increaseBy(int n)

    x increased by n

    ensures x == old(x) + n

Dates back to the 60-ies

Bob Floyd (1936 – 2001)

Tony Hoare (1934 -)

Hoare triples

Notation: {P}S{Q}

precondition

postcondition

HISTORY OF PROGRAM VERIFICATION

My thesis
(around 2000)

State-of-the-art

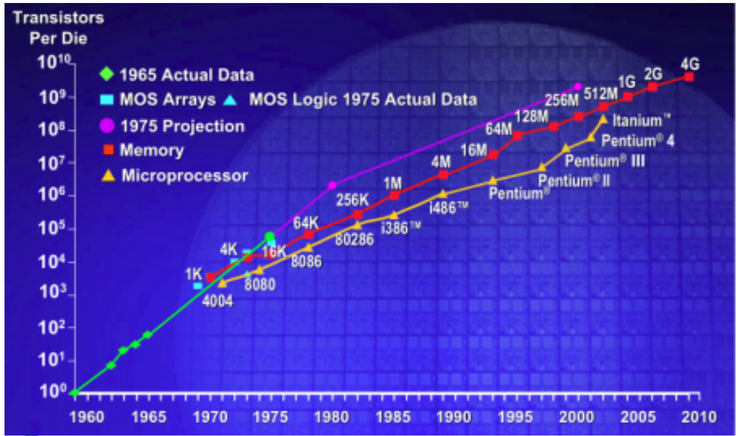
Floyd - Hoare

Dijkstra

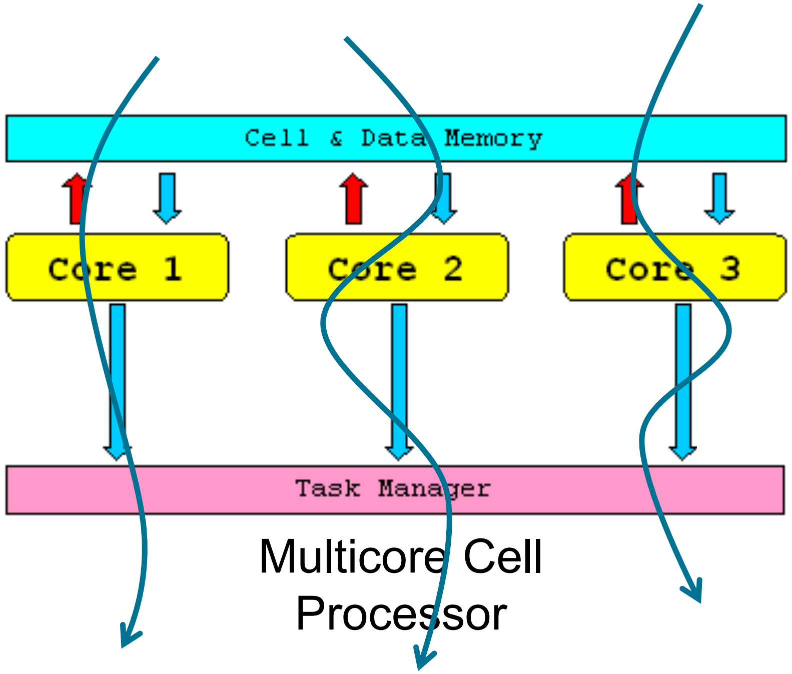
Krakatoa/Why3

THE FUTURE OF COMPUTING IS MULTICORE

Single core processors:
The end of Moore's law



Solution:
Multi-core processors

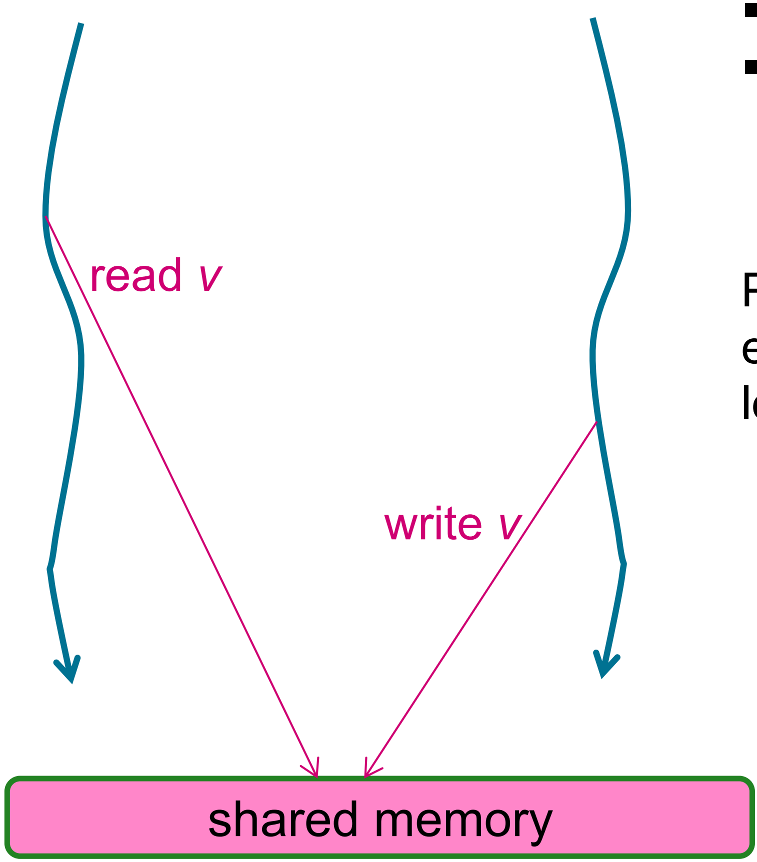


Multicore Cell Processor

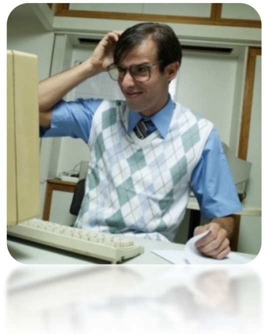
Multiple threads of execution

Coordination problem shifts from hardware to software

MULTIPLE THREADS CAUSE PROBLEMS



- Order?
- More threads?

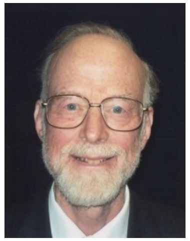


Possible consequences:
errors such as data races caused lethal bugs as in Therac-25

VERIFICATION OF MULTITHREADED PROGRAMS

Dijkstra

Floyd - Hoare



LOOP FOREVER

Code Contracts

JML

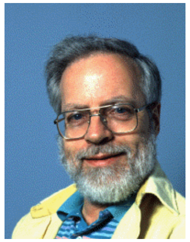
KeY

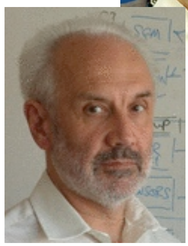
Dafny

Krakatoa/Why3

Concurrency (multithreading)

Owicki - Gries



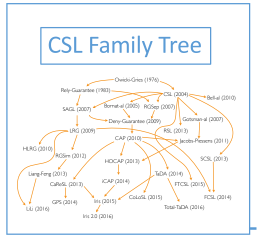


Jones

O'Hearn

2004
separation logic

VerCors



CARP

Silicon
Verifast

SPECIFICATIONS IN A CONCURRENT SETTING

requires true

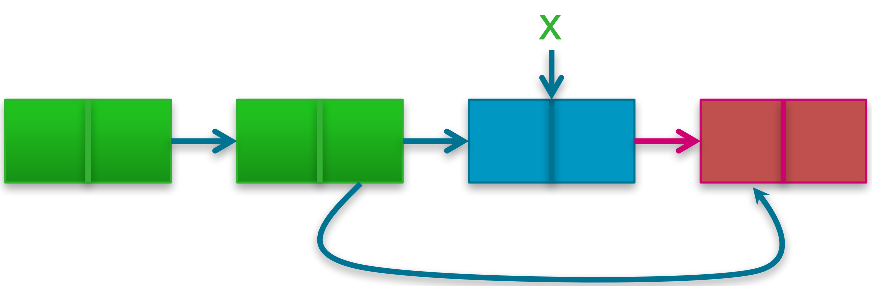
ensures x is the last element in the list

```
void addToList(Elem x) {
    // code
}
```

Any other thread might invalidate this!

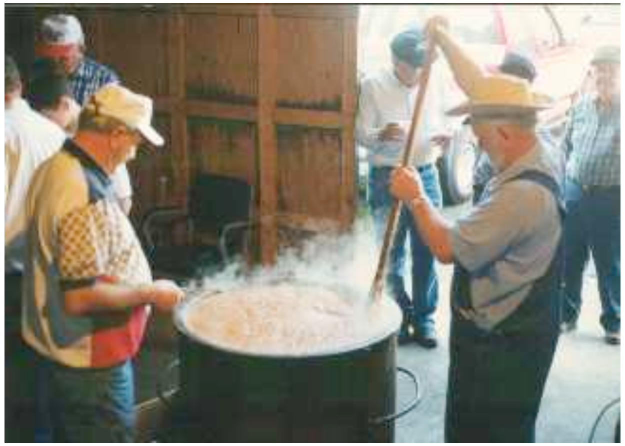
'x is in the list' cannot even be guaranteed!

Except when no other thread can update the list



RECIPE FOR REASONING ABOUT JAVA

- Separation logic for sequential Java (Parkinson)
- Concurrent Separation Logic (O'Hearn)
- Permissions (Boyland)



Permission-based Separation Logic for Java

SEPARATION LOGIC FOR JAVA

$$\overline{\{e.f \to \_\}\ e.f := v\ \{e.f \to v\}}$$

$$\overline{\{X = e \wedge X.f \to Y\}\ v := e.f\ \{X.f \to Y \wedge v = Y\}}$$

where X and Y are logical variables

Points-to permissions $e.f \to v$

- *e.f* contains *v*
- grants access to *e.f*

Matthew Parkinson

JOHN REYNOLDS'S 70TH BIRTHDAY PRESENT: CONCURRENT SEPARATION LOGIC

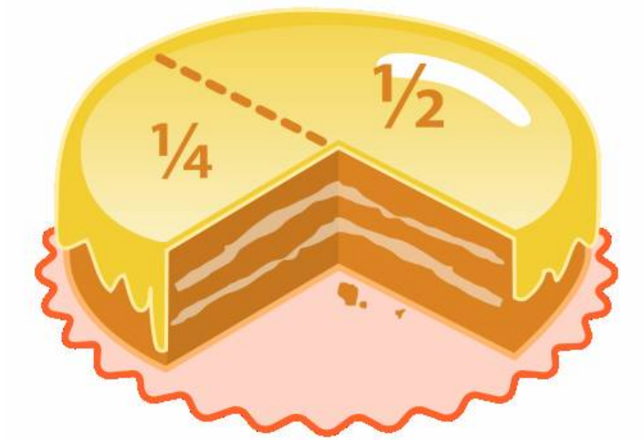
$$\frac{\{P1\}S1\{Q1\} \quad .... \quad \{Pn\}Sn\{Qn\}}{\{P1 \ast ... \ast Pn\}\ S1 \,||\, ... \,||\, Sn\ \{Q1 \ast ... \ast Qn\}}$$

where no variable free in *Pi* or *Qi* is changed in *Sj* (if $i \neq j$)

PERMISSIONS

John Boyland

- Permission to access a variable
- Value between 0 and 1
- Full permission 1 allows to change the variable
- Fractional permission in (0, 1) allows to inspect a variable
- Points-to predicate decorated with a permission
- Global invariant: for each variable, the sum of all the permissions in the system is never more than 1
- Permissions can be split and combined
- Thus: simultaneous reads allowed,
  but no read-write or write-write conflicts
  (data races)



VERCORS LOGIC

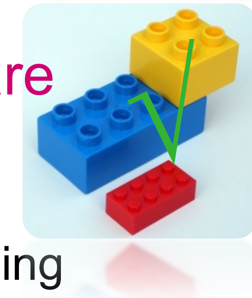
- Permission-based separation logic
- Rather than defining new logics, reuse existing verification technology

For convenience, we use implicit dynamic frames:
Perm(x,1/2) ** x == 4

Notation separating conjunction
- ** in textual representation, program annotations
- $\star$ in formal notation

VERCORS TECHNOLOGY

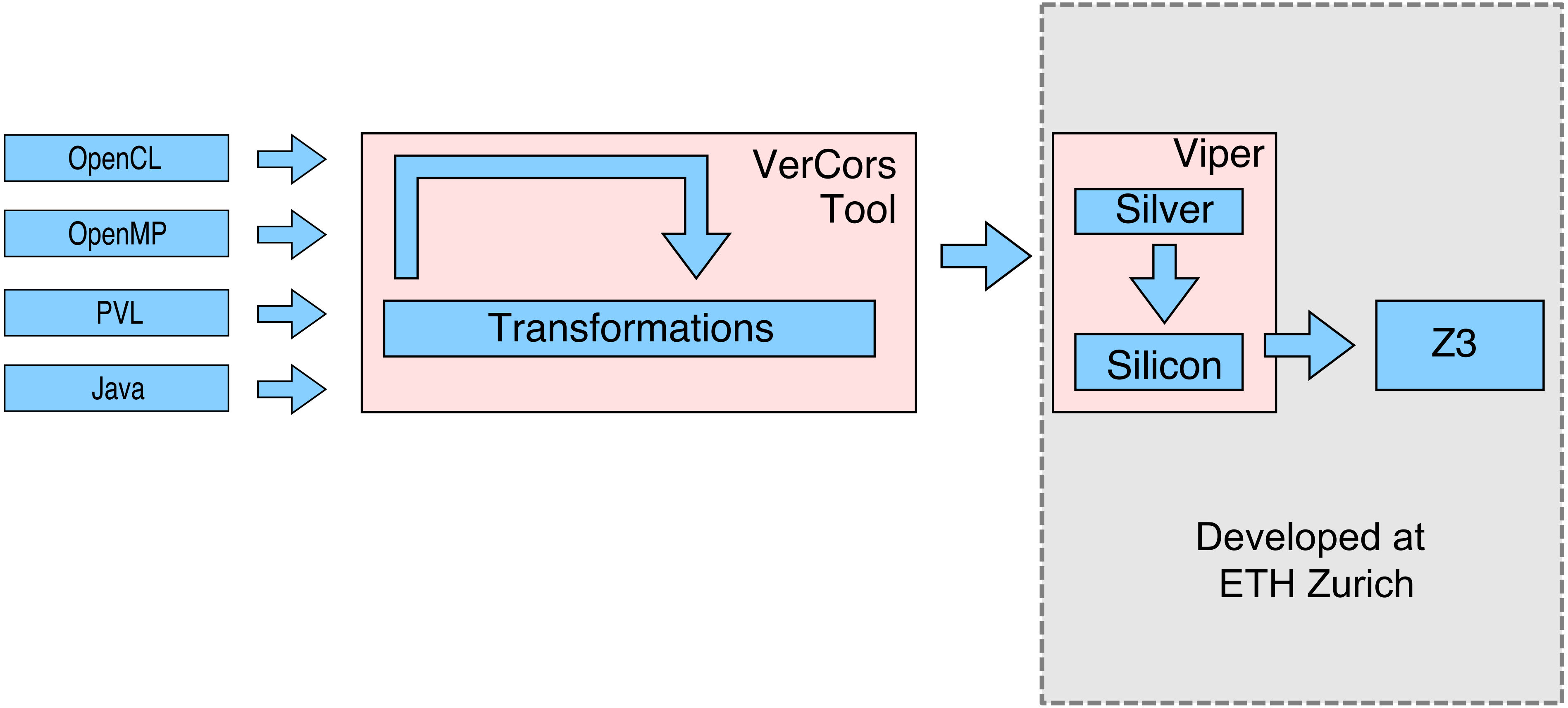
Automated verification of concurrent software



- Collection of verified concurrent data structures
- Generic verification theory of concurrent programming
  - Different concurrency and synchronisation techniques
  - Functional program properties
  - Different programming languages
  - Different concurrency paradigms
- Tool support
- To be continued:
  - Annotation generation
  - Automation

Mercedes
MAXIMAL RELIABILITY OF CONCURRENT AND DISTRIBUTED SOFTWARE

VERCORS TOOL ARCHITECTURE



A DIFFERENT APPROACH TO CONCURRENCY

- Parallel programming is difficult and error-prone
- In many cases concurrency is an optimization rather than intrinsic to the behavior of the program
- Intended behavior is often the same as sequential counterpart of the concurrent program

- Write a sequential program, and let the compiler parallellise it!

DETERMINISTIC PARALLEL PROGRAMMING

Sequential Program annotated by Compiler Directives

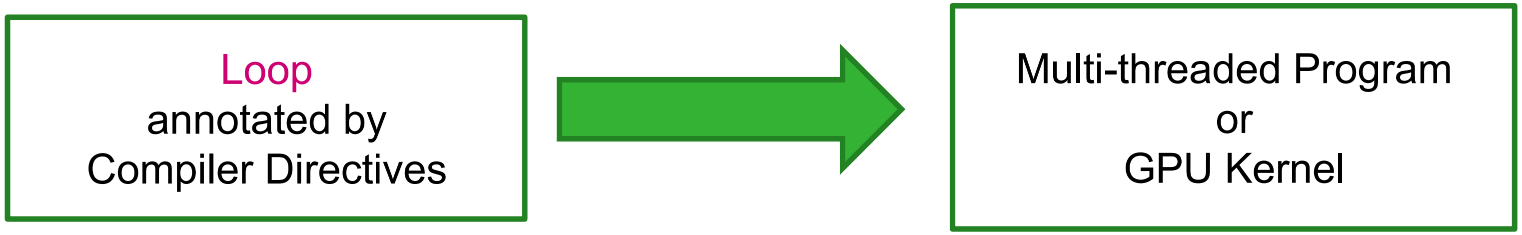
Compiled into →

Multi-threaded Program or GPU Kernel

- Compiler directives: hints to compiler to know where and how to parallelize sequential code
- Examples are ample: OpenMP, OpenACC, PENCIL, parallel_for constructs
- Homogeneous parallellism

BACKGROUND: LOOP PARALLELLISATIONS

Earlier work (FASE 2015): how to reason about the correctness of Loop Parallelisations




```
#pragma independent
for(int i=0; i < N; i++){
  S1: a[i]=a[i]*CONSTANT+c[i]*(1-CONSTANT);
  S2: c[i]=min(a[t[i]],MAX_VALUE);    }
```

for t[] ={1, 2, 3, …}

```
Iteration 1
S1: a[1] = a[1]*CONSTANT + c[1]*(1- CONSTANT);
S2: c[1] = min(a[2],MAX_VALUE);
```

```
Iteration 2
S1: a[2] = a[2]*CONSTANT + c[2]*(1- CONSTANT);
S2: c[2] = min(a[3],MAX_VALUE);
```

BACKGROUND: ITERATION CONTRACT

Specify a contract for every iteration of a loop.

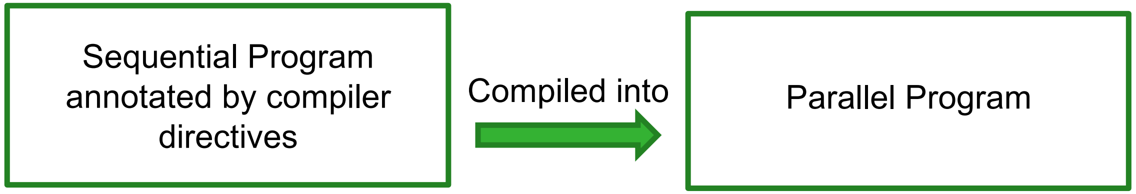
- Simpler than classical loop invariants
- Parallellisability of loop follows
- Both data race freedom and functional specifications

Example

```
/*@  requires Perm(a[i],1) ** Perm(b[i],1/2);
     ensures Perm(a[i],1) ** Perm(b[i],1/2);  @*/
  #pragma independent
  for(int i=0; i < N; i++)
    { S1: a[i]= 2 * b[i]; }
```

- Method can account for dependences too:
  - loop vectorization
  - reductions

OPENMP COMPILER DIRECTIVES



OpenMP Examples

```
…
#pragma omp parallel {
#pragma omp for
for(int i =0;i<N;i++)
    c[ i ] =a[i ];
#pragma omp for
for(int i =0;i<N;i++)
    d[ i ]=c[i+1]+b[i ];
} …
```

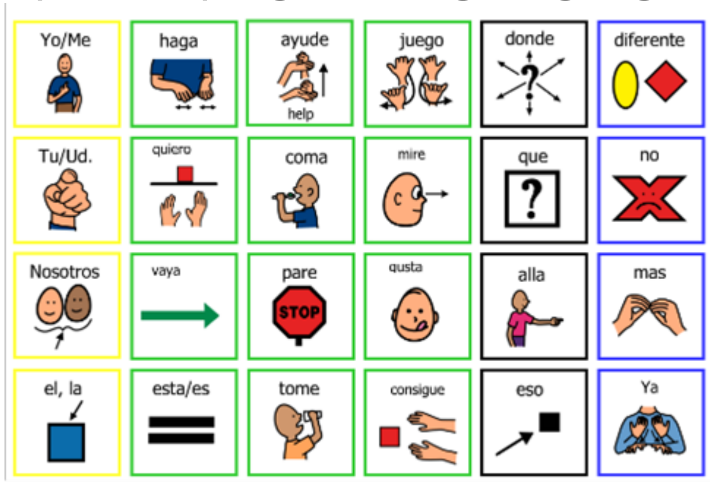
```
…
#pragma omp parallel {
#pragma omp for schedule(static) nowait
for(int i =0;i<N;i++)
    c[ i ] = a[i ];
#pragma omp for schedule(static) nowait
for(int i =0;i<N;i++)
    d[ i ] = c[i ]+b[i ];
} …
```

```
…
#pragma omp parallel {
#pragma omp sections
{
#pragma omp section
#pragma omp parallel for
    for(int i =0;i<N;i++)
      c[ i ]=a[i ];
#pragma omp section
#pragma omp parallel for
    for(int i =0;i<N;i++)
      d[ i ]=d[i ]+b[i ];
} …
```

APPROACH

- Define a language to capture the exact semantics of these OpenMP annotations (with precise semantics)
- Develop verification technique for core language
- Encode OpenMP into this core language

- Future: encode more deterministic parallel programming languages into core language



PARALLEL PROGRAMMING LANGUAGE: PPL

Observation

OpenMP annotations define the code blocks and their composition

```
#pragma omp parallel {
#pragma omp for
for(int i =0;i<N;i++)
    c[ i ] = a[ i ];
#pragma omp for
for(int i =0;i<N;i++)
    d[ i ] = c[ i+1 ] + b[ i ];
}
```

Par(N) body1(tid);

⨟

Par(N) body2(tid);

Sequential composition of parallel blocks

```
#pragma omp parallel {
#pragma omp for schedule(static) nowait
for(int i =0;i<N;i++)
    c[ i ] = a[i ];
#pragma omp for schedule(static) nowait
for(int i =0;i<N;i++)
    d[ i ] = c[ i ] + b[i ];
}
```

Par(N) body1(tid);

$\oplus$

Par(N) body2(tid);

Fusion of parallel blocks

```
#pragma omp parallel {
#pragma omp sections
{
 #pragma omp section
#pragma omp parallel for
 omp_for(int i =0;i<N;i++)
    c[ i ] = a[i ];
 #pragma omp section
#pragma omp parallel for
omp_ for(int i =0;i<N;i++)
    d[ i ] = d[i ] + b[i ];
}}
```

Par(N) body1(tid);

||

Par(N) body2(tid);

Parallel composition of parallel blocks

SYNTAX OF PPL

Paper: formal semantics for full language

PPL: a core language for deterministic parallel programming

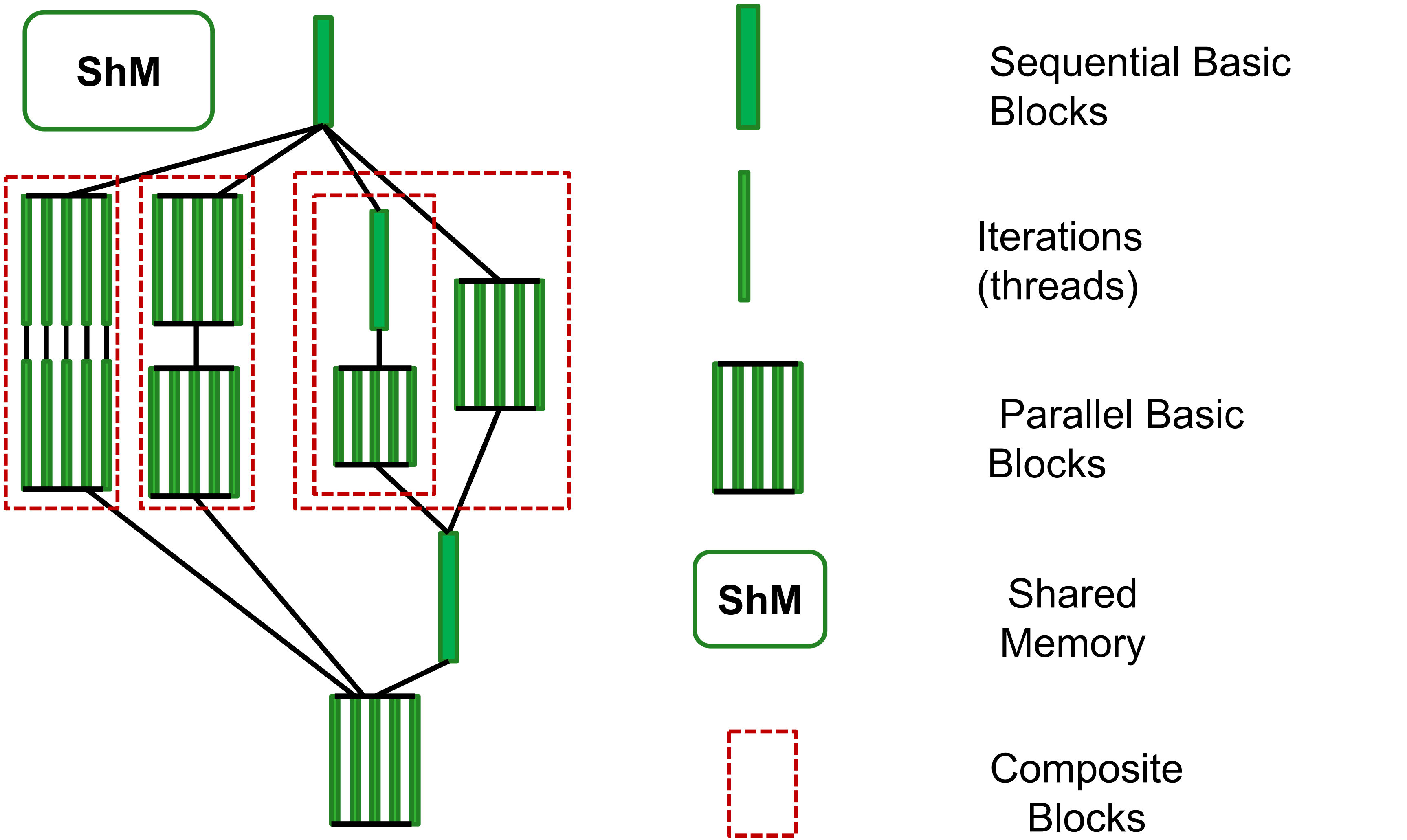
Block Compositions

$$
\begin{array}{lcl}
\mathsf{Block} & ::= & (\mathsf{Block} \mid\mid \mathsf{Block}) \mid (\mathsf{Block} \oplus \mathsf{Block}) \mid (\mathsf{Block} \fatsemi \mathsf{Block}) \mid \mathsf{Par(N)}\ \mathsf{S} \mid \mathsf{S} \\
\mathsf{S} & ::= & s;\mathsf{S} \mid \mathsf{skip} \\
s & ::= & \mathsf{ass} \mid \mathsf{if}\ (b)\ \{\mathsf{S}\}\ \mathsf{else}\ \{\mathsf{S}\} \mid \mathsf{while}\ (b)\ \{\mathsf{S}\} \mid \mathsf{Vec(N)}\ \mathsf{V} \\
\mathsf{V} & ::= & b \Rightarrow \mathsf{ass};\mathsf{V} \mid \mathsf{skip} \\
\mathsf{ass} & ::= & v := e \mid v := \mathsf{mem}(e) \mid \mathsf{mem}(e) := v \\
b & ::= & \text{boolean expression over private memory} \\
e & ::= & \text{arithmetic expression over private memory} \\
v & ::= & \text{thread local variable}
\end{array}
$$

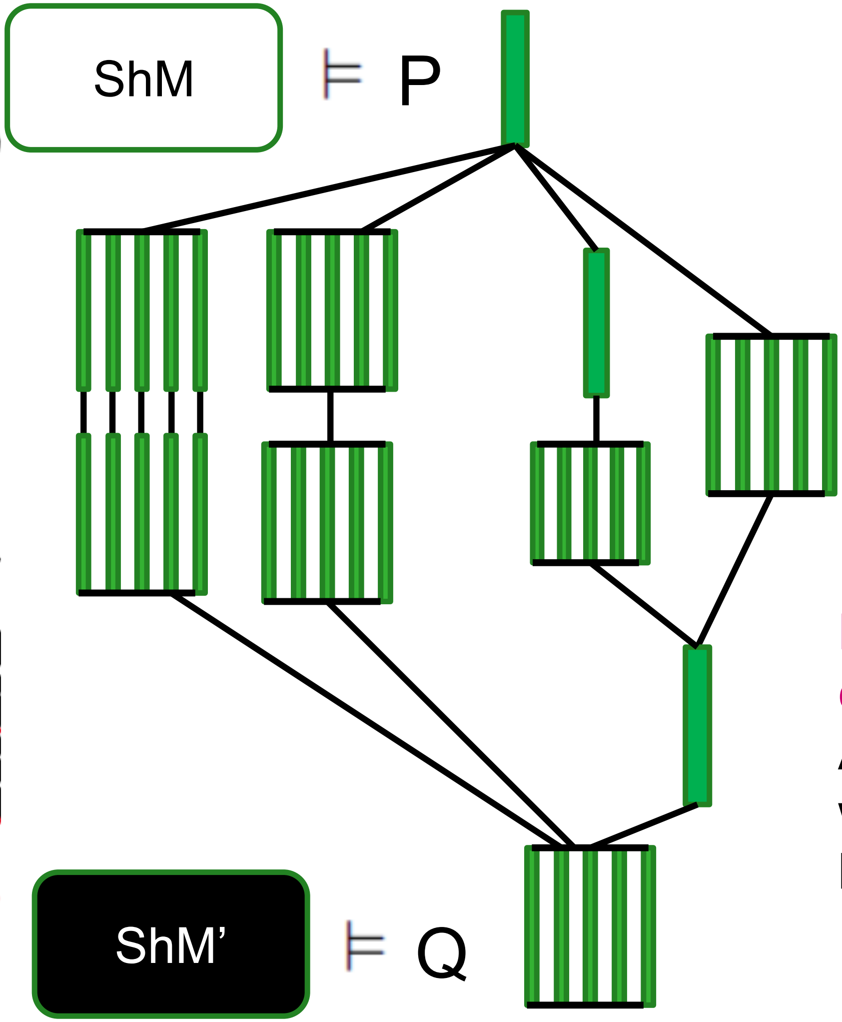
Parallel and Sequential basic blocks

SHAPE OF PPL PROGRAMS

THE FORK-JOIN MODEL



VERIFICATION OF PPL PROGRAMS



Problem1: Data race freedom

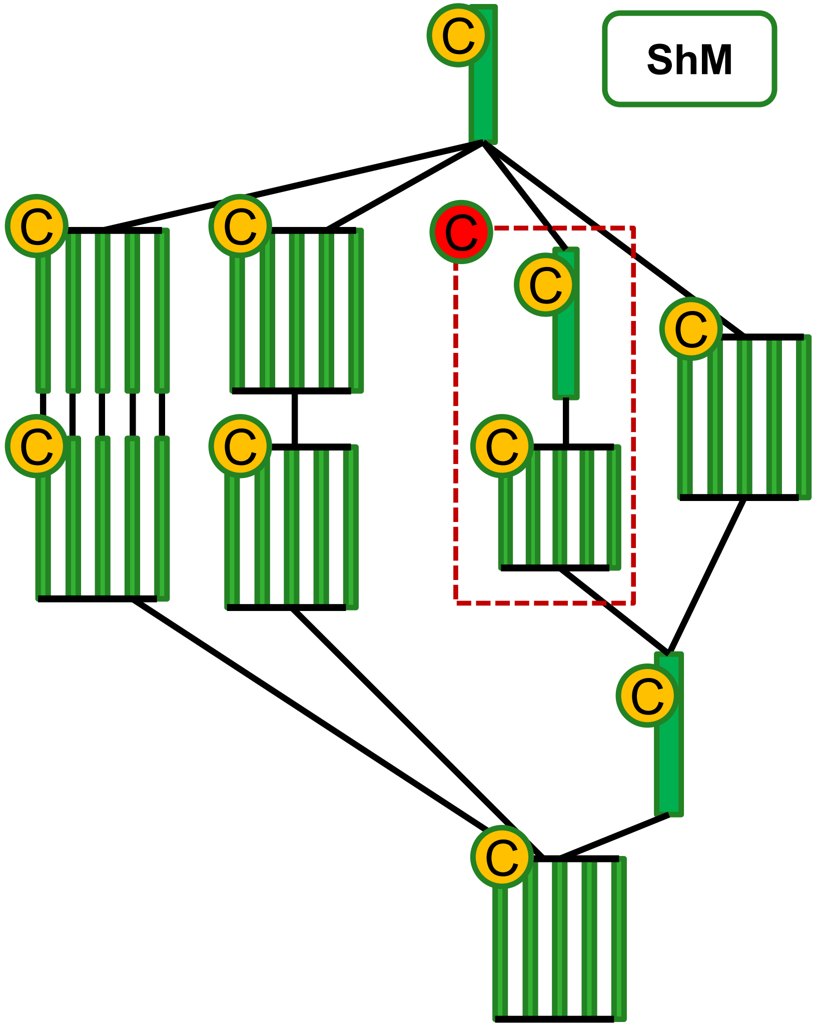
- No two (parallel) threads have racy access (read/write or write/write) to the shared memory
- Implies correctness of high-level annotations (e.g. OpenMP annotations)

Problem 2: Functional correctness

A verified PPL program is correct w.r.t. its specified functional behavior:

{P} PPL-Prog {Q}

DATA RACE FREEDOM



Solution:

1. Verify all basic blocks in isolation w.r.t iteration contract

Iteration Contract C

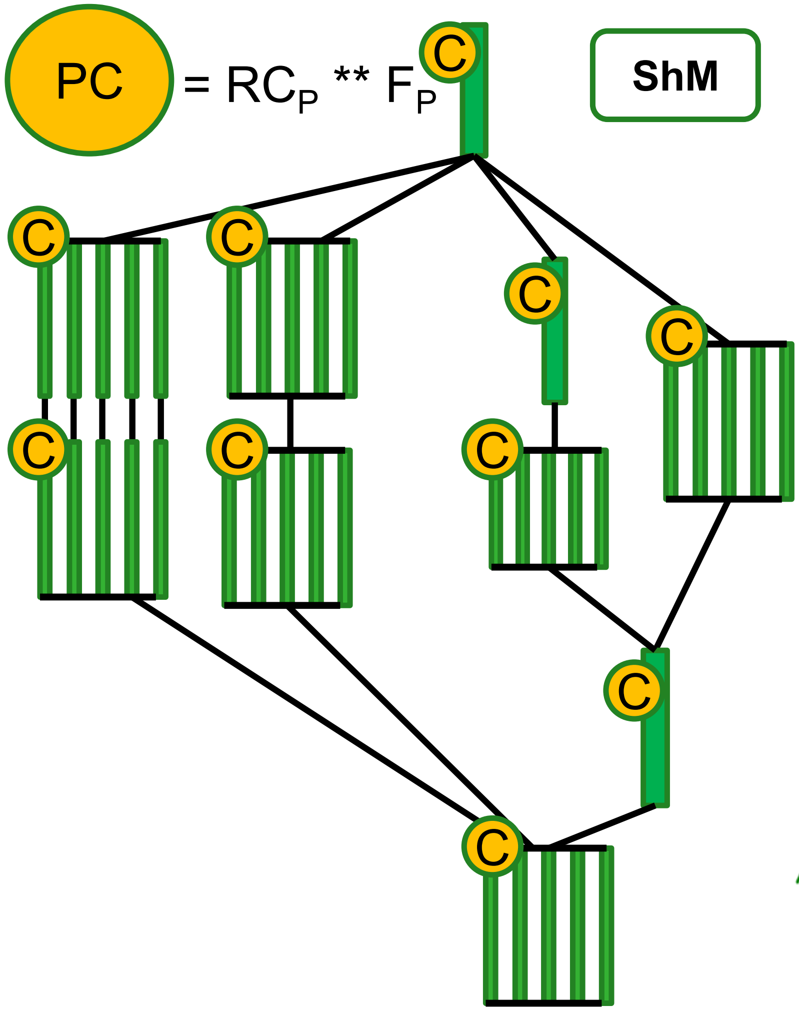
```
for(int i=0; i < N; i++)
/*@ requires Perm(a[i],1) ** Perm(b[i],1/2);
    ensures Perm(a[i],1) ** Perm(b[i],1/2);  @*/
 { S1: a[i]= 2 * b[i]; }
```

2. Verify block compositions

Requires extra contracts for composite blocks ☹….

However, iteration contracts are sufficient to prove data race freedom of whole program

DATA RACE FREEDOM – COMPOSITE BLOCKS



Observation

PPL program is a partial order over the set of all iterations under happens-before relation

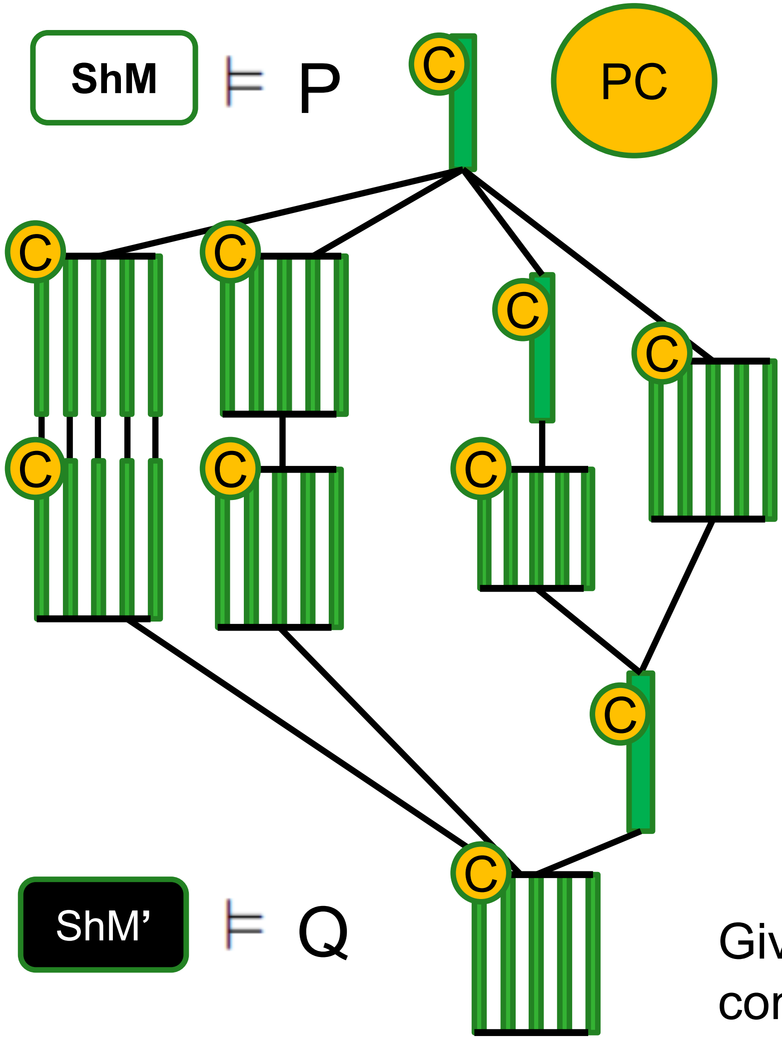
Set of independent iteration pairs $\mathfrak{I}_{\perp}^{\mathcal{P}}$:
All iterations (from different basic blocks) which are not ordered under happens-before relation

Block composition is correct if

$$\forall (i^{b}, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.(RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j))$$

```
for(int i=0; i < N; i++)
/*@ requires Perm(a[i],1) ** Perm(b[i],1/2);
    ensures Perm(a[i],1) ** Perm(b[i],1/2);  @*/
  { S1: a[i]= 2 * b[i]; }
```

FUNCTIONAL CORRECTNESS



Solution:

Extend Iteration Contracts with functional properties:

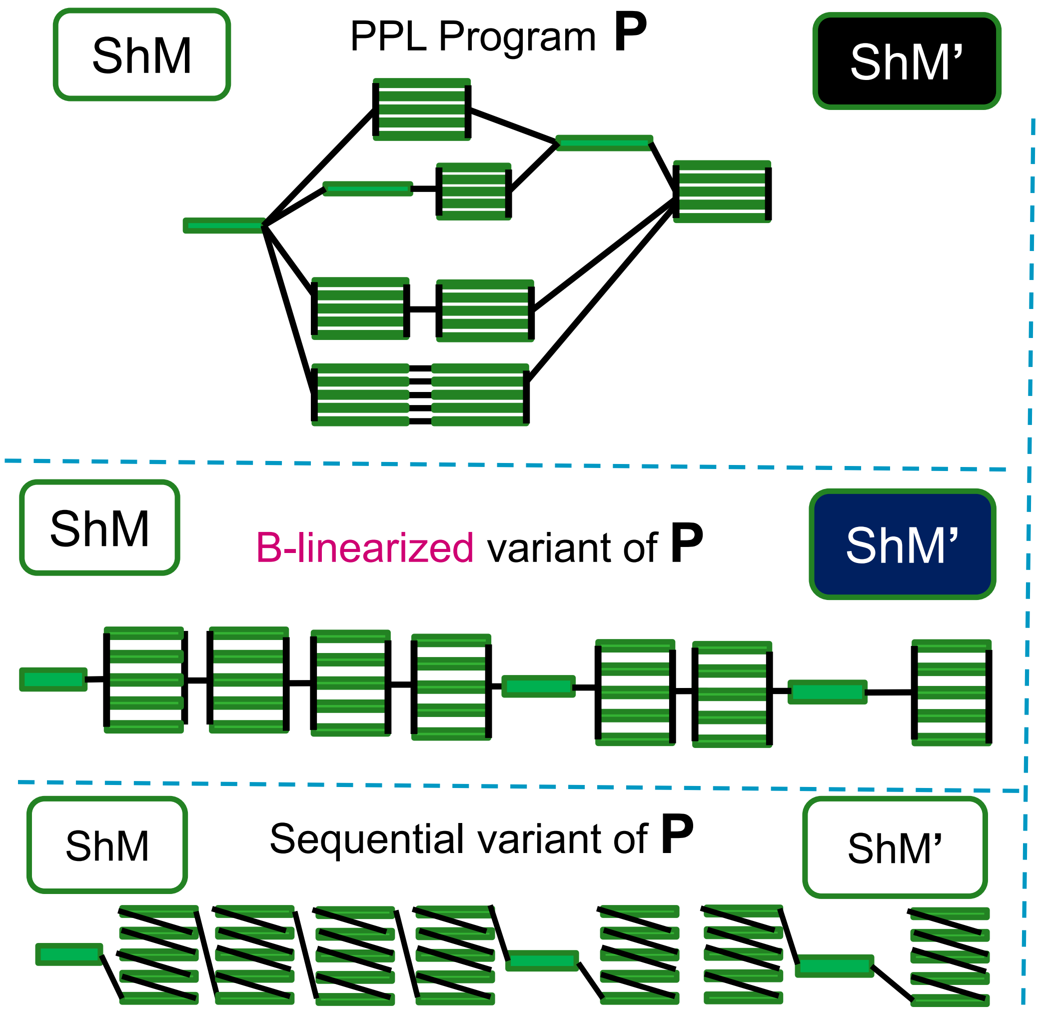
C ≡ R ★ F

Example:

```
for(int i =0;i<N;i++)
/*@ requires Perm(a[i],1) ** Perm(b[i],1/2)
    requires b[i] =1;
    ensures Perm(a[i],1) ** Perm(b[i],1/2)
    ensures  a[i] = 2;
@*/
      {a[ i ] = b[ i ] * 2;}
```

Given **P** and all iteration contracts, can we conclude **Q**?

FUNCTIONAL CORRECTNESS: B-LIN REDUCTION



Data race freedom of block compositions:

$$\forall (i^b, j^{b'}) \in \mathfrak{I}^{\mathcal{P}}_{\perp}.(RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j))$$

implies

ShM' = ShM'

Data race freedom of basic blocks

w.r.t to their iteration contracts (FASE 2015)

ShM' = ShM'

VERIFICATION OF PPL PROGRAMS

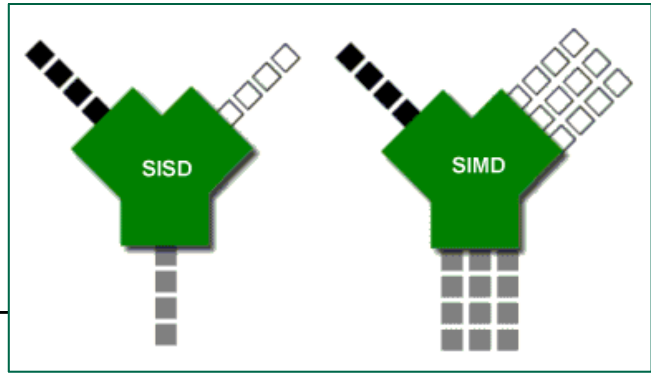
Verify all basic blocks in isolation w.r.t. iteration contract

Data race freedom
Parallelisation is correct if block composition is correct

Functional correctness
If parallelization is correct, then the behavior of PPL program is equivalent to the behavior of its sequential counterpart.

$$\frac{\left(\forall (i^{b}, j^{b'}) \in \mathfrak{I}^{\mathcal{P}}_{\perp}.(RC_{\mathcal{P}} \rightarrow rc_{b}(i) \star rc_{b'}(j))\right) \quad \{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} blin(\mathcal{P}) \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}}{\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} \mathcal{P} \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}} \text{[b–linearise]}$$

DATA PARALLELLISM

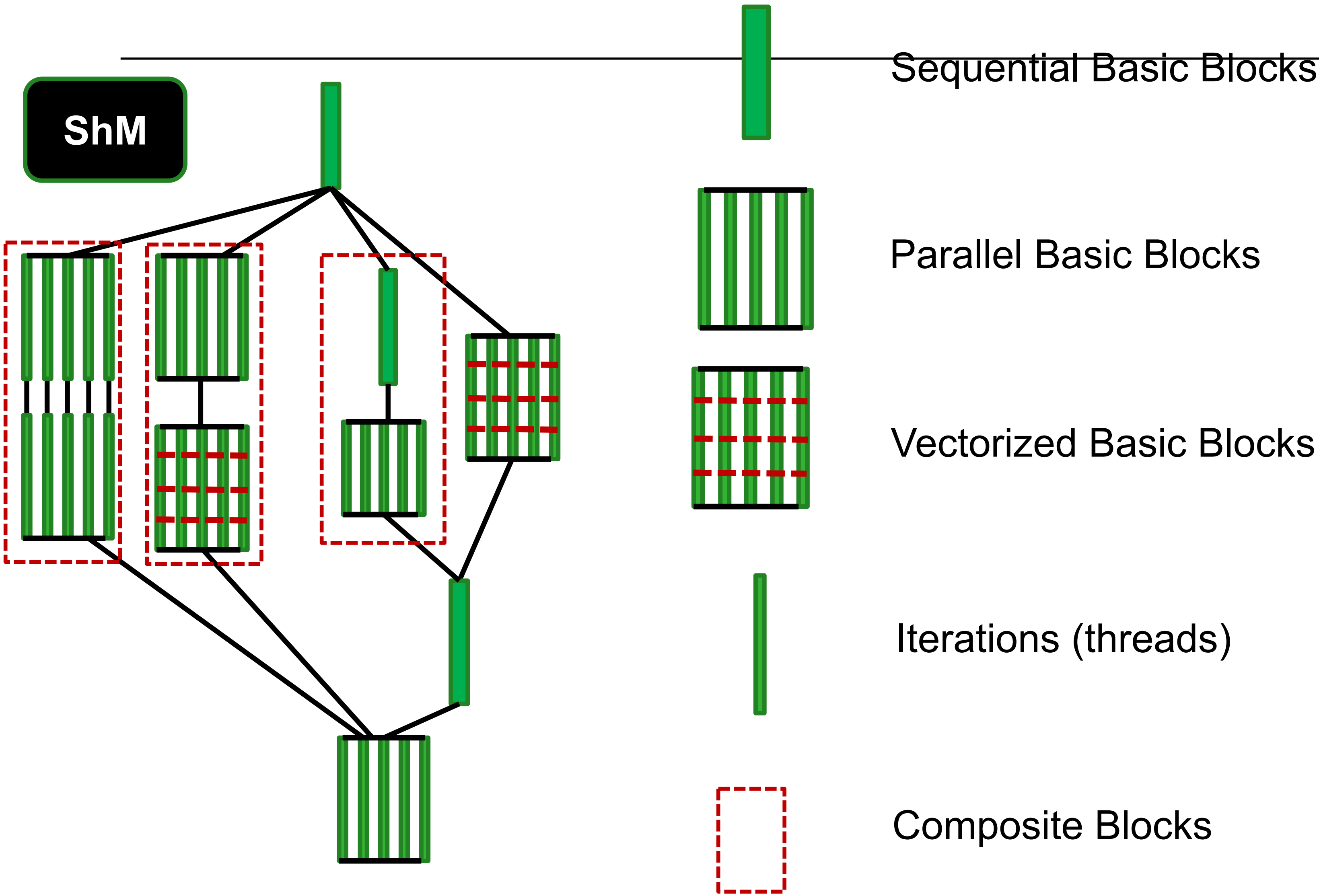


- SIMD: Single Instruction Multiple Data
- Also known as vector instructions
- SIMD support from OpenMP 4.0 with simd and for simd annotations where loop body executes in lock-step fashion.

```
#pragma omp parallel for simd  simdlen(M)
 for(int i =0;i<N;i++) {
    c[ i ]=a[i] +2;
    ---------------------
    d[i] = c[i] *b[i];
 }
```

```
#pragma omp simd  simdlen(M)
 for(int i =0;i<N;i++) {
    c[ i ]=a[ i ] +2;
    ---------------------
    d[i] = c[ i+M ] *b[i];
 }
```

SHAPE OF PPL PROGRAMS WITH SIMD



LOOP WITH FORWARD DEPENDENCE

```
for(int i=0; i < N; i++)
/*@ requires Perm(a[i],1) ** Perm(b[i],1/2) ** Perm(c[i],1);
    ensures Perm(b[i],1/2) ** Perm(a[i],1/2) ** Perm(c[i],1);
      ensures i>0 ==> Perm(a[i−1],1/2);
      ensures i==N−1 ==> Perm(a[i],1/2);
@*/
{ a[i]=b[i]+1;
  //@ L1:if (i< N−1) send Perm(a[i],1/2) to L2,1;
  //@ L2:if (i>0) recv Perm(a[i−1],1/2) from L1,1;
  if (i>0) c[i]=a[i−1]+2; }
```

Send/receive:

- Transfer permissions to different iteration
- Needed for verification of this iteration contract
- Send indicates the necessary synchronisation

Challenge: identify the independent iterations pairs

CONCLUSIONS

- VerCors project: verification of concurrent software using permission-based separation logic as specification language
- Formalizing PPL: a core language for deterministic parallel programming
- A verification technique for reasoning about data race freedom and functional correctness of PPL program
- Soundness of our approach is proven
- Enabling verification of OpenMP programs via encoding into PPL
- Investigating how to address state-of-the-art features of OpenMP such as simd loops

FUTURE WORK

- Extend approach to handle intra-block data dependencies which requires permission transfer between the iterations of the loop
  - Including more complicated cases of for-simd in OpenMP
- Supporting atomic operations and reductions in OpenMP
- Supporting OpenMP tasks
- Encoding other DPP languages (e.g. Cilk) into PPL
- Automatic generation of iteration contracts

THE END…

Automated verification of
concurrent software

More information and try the tool:
http://www.utwente.nl/vercors