Multi-threaded ASP Solving with *clasp* 2

Martin Gebser Benjamin Kaufmann Torsten Schaub

University of Potsdam

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Introduction

- Goal Leverage the power of today's multi-core machines for supporting parallel conflict-driven solving
- Approach Coarse-grained, task-parallel approach via shared memory multi-threading
- Result *clasp 2* allows for parallel ASP, PB, and SAT solving via search space splitting and/or competing strategies

Introduction

- Goal Leverage the power of today's multi-core machines for supporting parallel conflict-driven solving
- Approach Coarse-grained, task-parallel approach via shared memory multi-threading
- Result *clasp 2* allows for parallel ASP, PB, and SAT solving via search space splitting and/or competing strategies

Introduction

- Goal Leverage the power of today's multi-core machines for supporting parallel conflict-driven solving
- Approach Coarse-grained, task-parallel approach via shared memory multi-threading
- Result *clasp 2* allows for parallel ASP, PB, and SAT solving via search space splitting and/or competing strategies

Conflict-Driven Answer Set Solving

- Approach Computation of answer sets of logic programs, based on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Checking (SAT)
- Idea View inferences in Answer Set Programming (ASP) as unit propagation on nogoods
- Benefits
  - A uniform constraint-based framework for different kinds of inferences in ASP
  - Advanced techniques from the areas of CP and SAT
  - Highly competitive implementation
- Awards *clasp* won several prizes at ASP, PB, and SAT competitions

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Towards conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  (DPLL stands for 'Davis-Putnam-Logemann-Loveland')
  - (Unit) propagation
  - (Chronological) backtracking
  - in ASP, eg *smodels*
- Modern CDCL-style approach
  (CDCL stands for 'Conflict-Driven Constraint Learning')
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
  - in ASP, eg *clasp*

DPLL-style solving

```
loop
    propagate                          // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                    // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            backtrack                  // unassign literals made after last decision
            flip                       // assign complement of last decision literal
```

CDCL-style solving

```
loop
    propagate                          // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                    // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            analyze                    // analyze conflict and add conflict constraint
            backjump                   // unassign literals until conflict constraint is unit
```

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Parallel CDCL-style solving in *clasp* 2

```
while work available
  while no (result) message to send
    communicate          // exchange information with other solver
    propagate                    // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide              // non-deterministically assign some literal
    else
        if root-level conflict then send unsatisfiable
        else if external conflict then send unsatisfiable
        else
            analyze              // analyze conflict and add conflict constraint
            backjump     // unassign literals until conflict constraint is unit
  communicate                    // exchange results (and receive work)
```

Parallel CDCL-style solving in *clasp* 2

```
while work available
  while no (result) message to send
    communicate          // exchange information with other solver
    propagate                    // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide              // non-deterministically assign some literal
    else
        if root-level conflict then send unsatisfiable
        else if external conflict then send unsatisfiable
        else
            analyze              // analyze conflict and add conflict constraint
            backjump     // unassign literals until conflict constraint is unit
  communicate                    // exchange results (and receive work)
```

Parallel CDCL-style solving in *clasp* 2

```
while work available
  while no (result) message to send
    communicate          // exchange information with other solver
    propagate                    // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide              // non-deterministically assign some literal
    else
        if root-level conflict then send unsatisfiable
        else if external conflict then send unsatisfiable
        else
            analyze              // analyze conflict and add conflict constraint
            backjump     // unassign literals until conflict constraint is unit
  communicate                    // exchange results (and receive work)
```

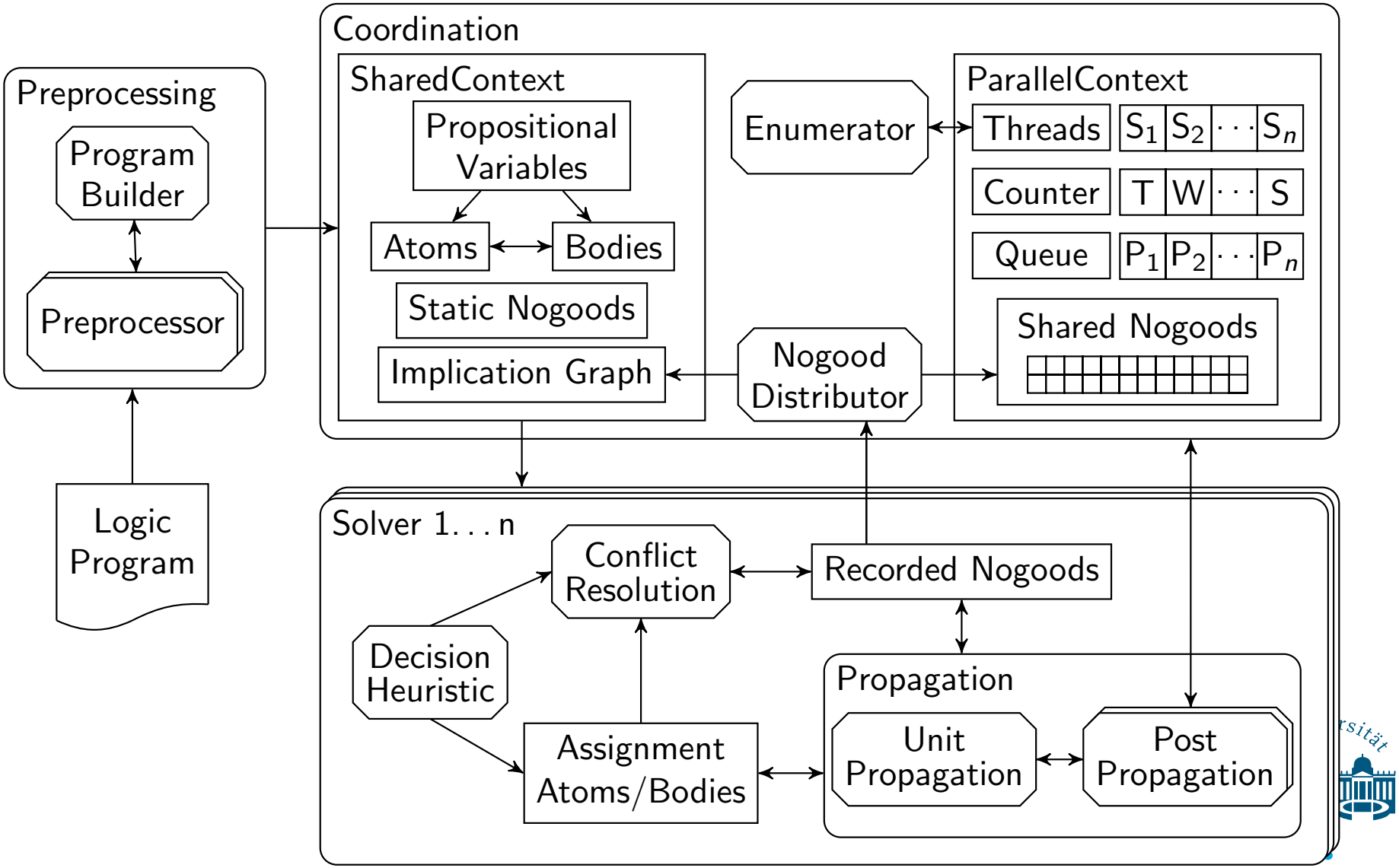
Parallel CDCL-style solving in *clasp* 2

```
while work available
  while no (result) message to send
    communicate          // exchange information with other solver
    propagate            // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide      // non-deterministically assign some literal
    else
        if root-level conflict then send unsatisfiable
        else if external conflict then send unsatisfiable
        else
            analyze      // analyze conflict and add conflict constraint
            backjump     // unassign literals until conflict constraint is unit
  communicate            // exchange results (and receive work)
```

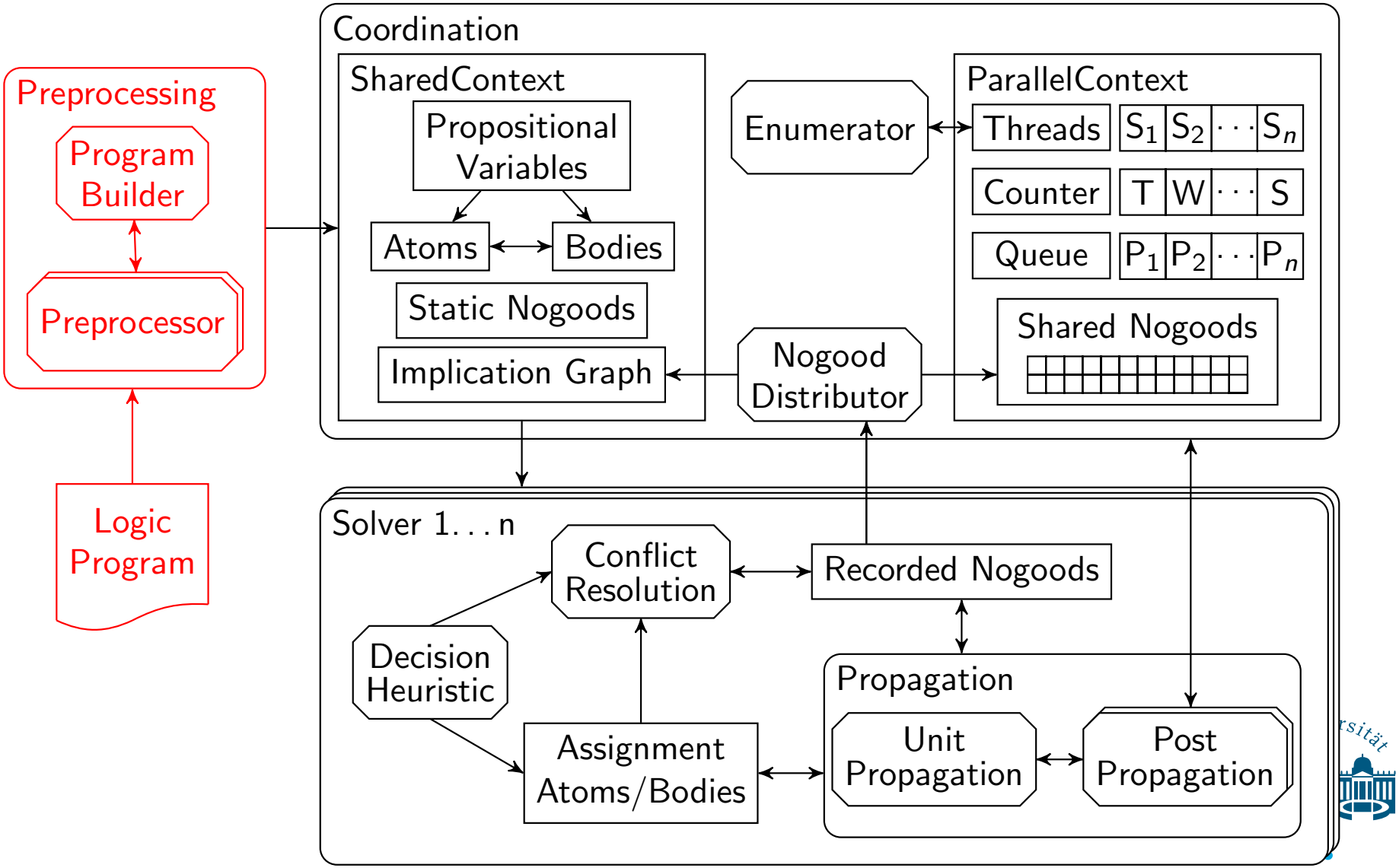
Parallel CDCL-style solving in *clasp* 2

```
while work available
  while no (result) message to send
    communicate          // exchange information with other solver
    propagate                    // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide              // non-deterministically assign some literal
    else
        if root-level conflict then send unsatisfiable
        else if external conflict then send unsatisfiable
        else
            analyze              // analyze conflict and add conflict constraint
            backjump     // unassign literals until conflict constraint is unit
  communicate                    // exchange results (and receive work)
```

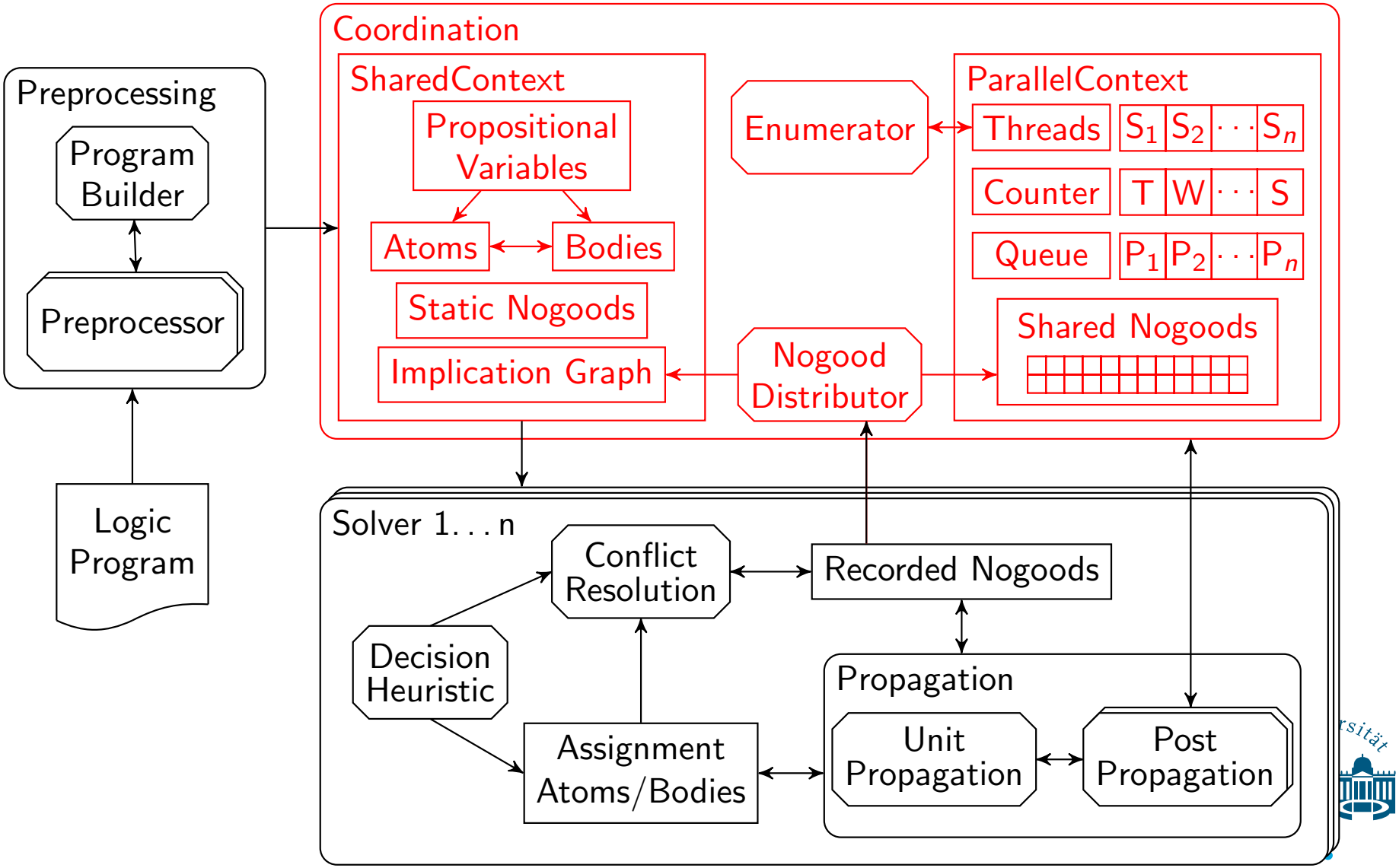
Multi-threaded architecture of *clasp*



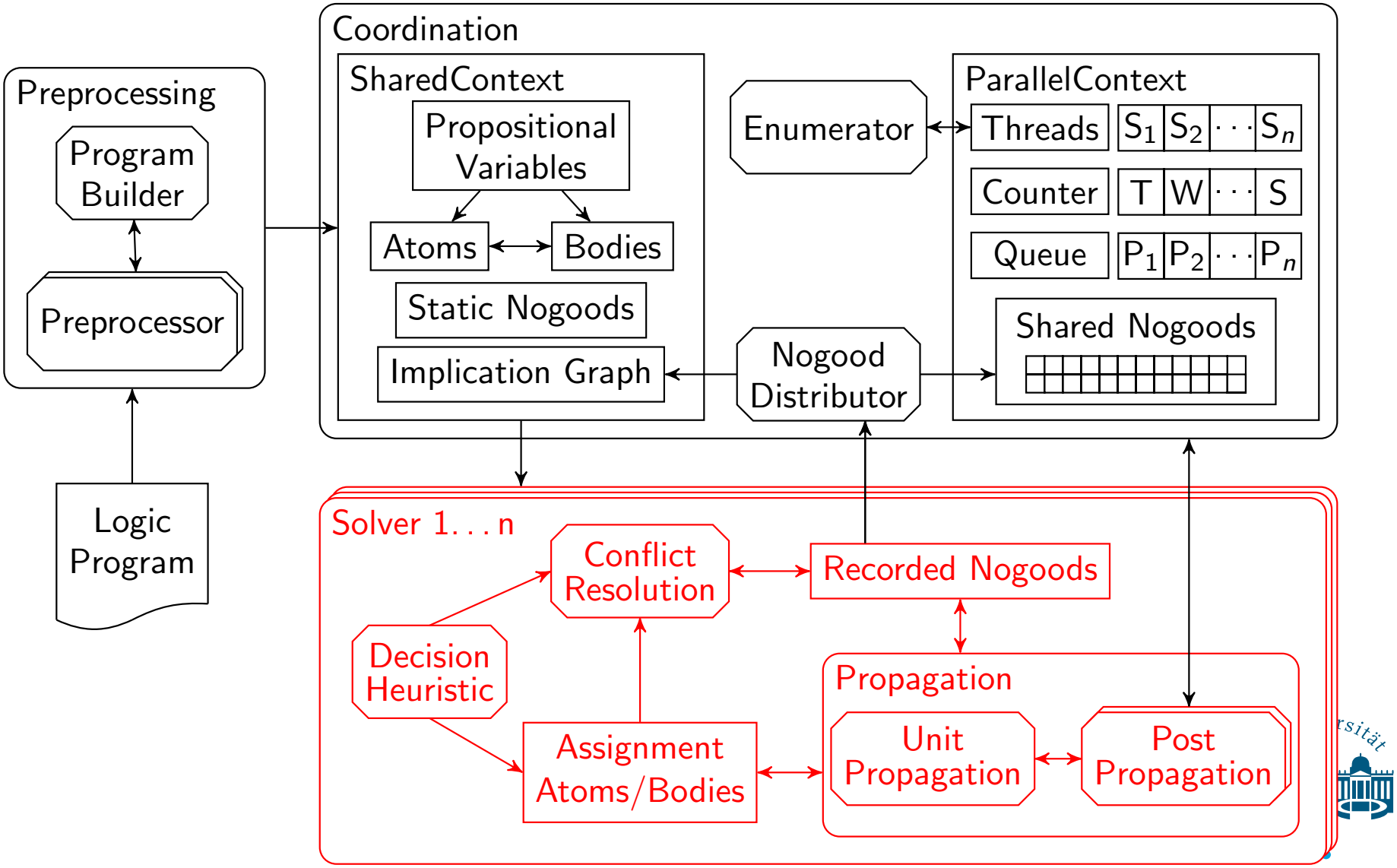
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Outline

1 Introduction

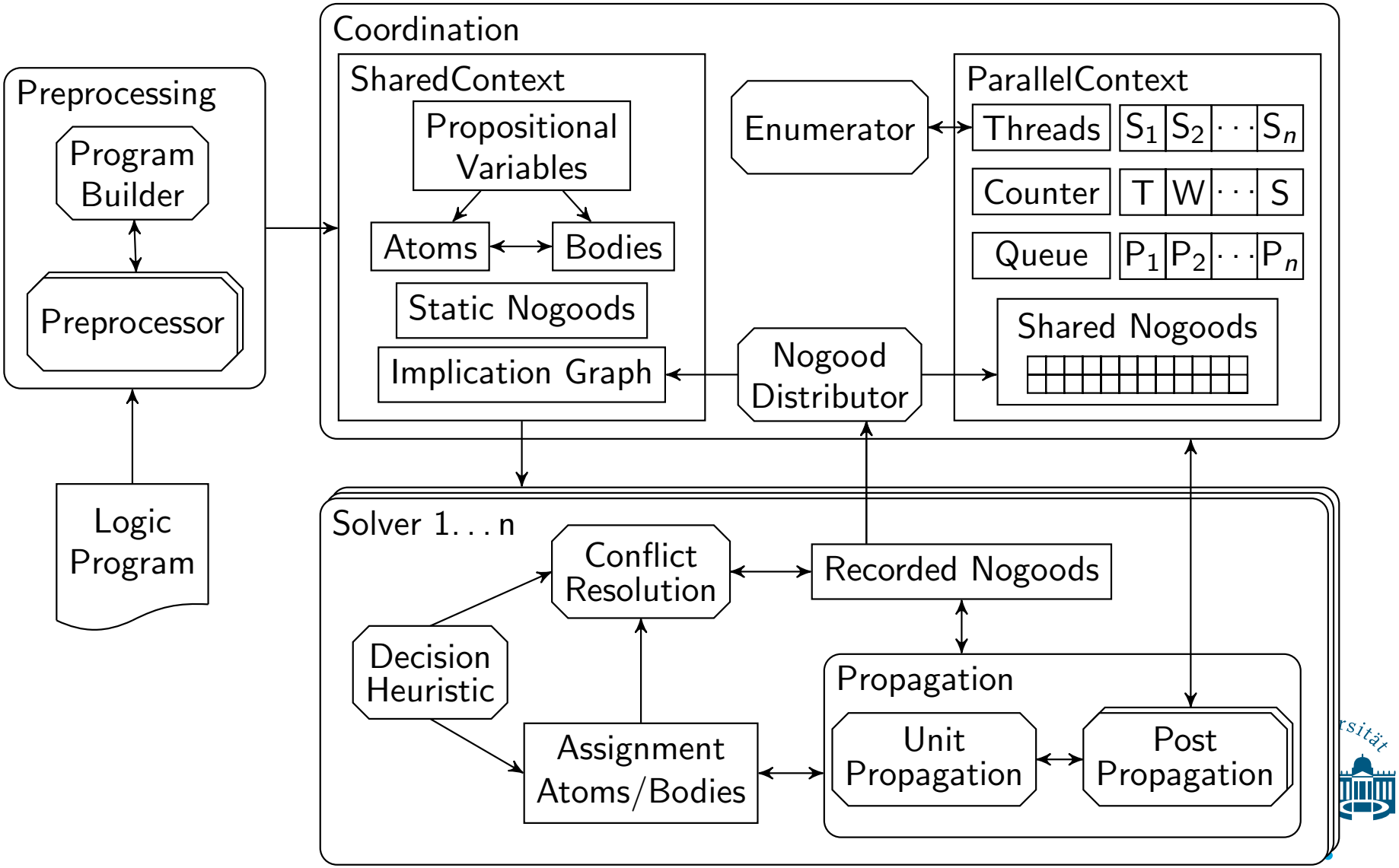
2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

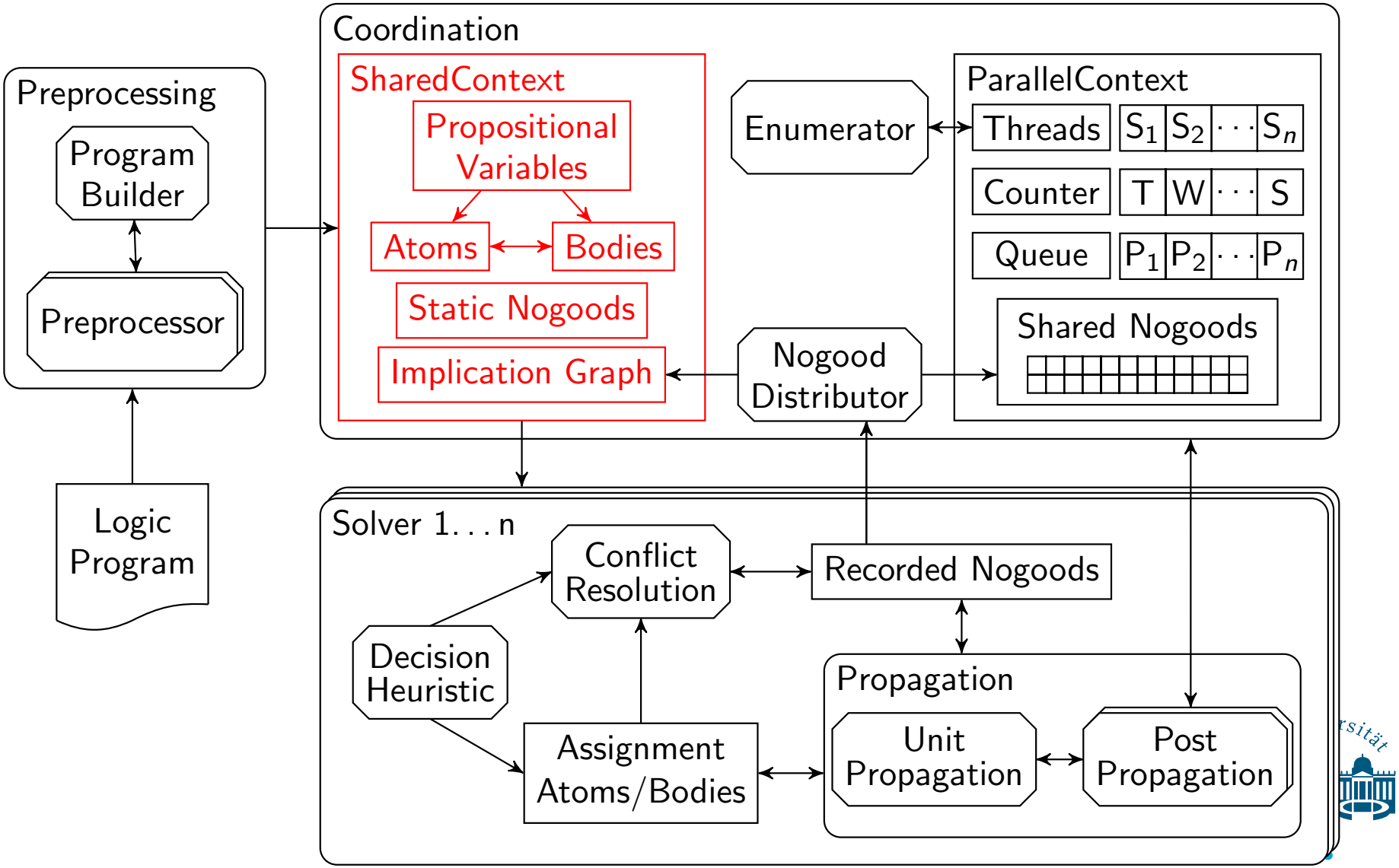
4 Experiments

5 Summary

Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



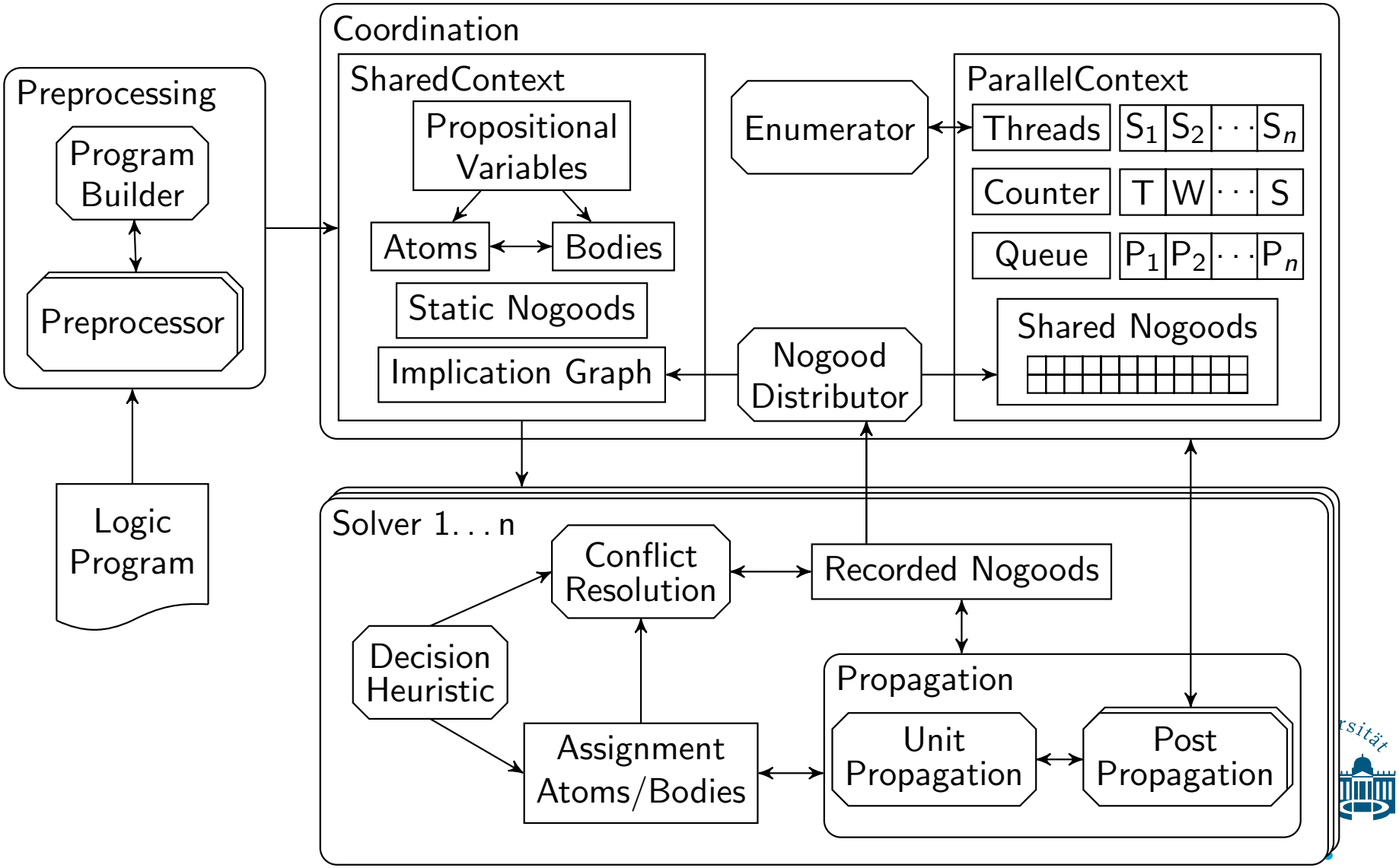
SharedContext

- The *SharedContext* object is initialized by the main thread and shared among all participating threads
- Among others, the *SharedContext* object contains
  - the set of relevant Boolean variables together with type information (eg atom, body, aggregate, etc),
  - a symbol table, mapping (named) atoms from the program to internal variables,
  - the positive atom-body dependency graph, restricted to its strongly connected components,
  - the set of Boolean constraints, among them nogoods, cardinality and weight constraints, minimize constraints, and
  - an implication graph capturing inferences from binary and ternary nogoods

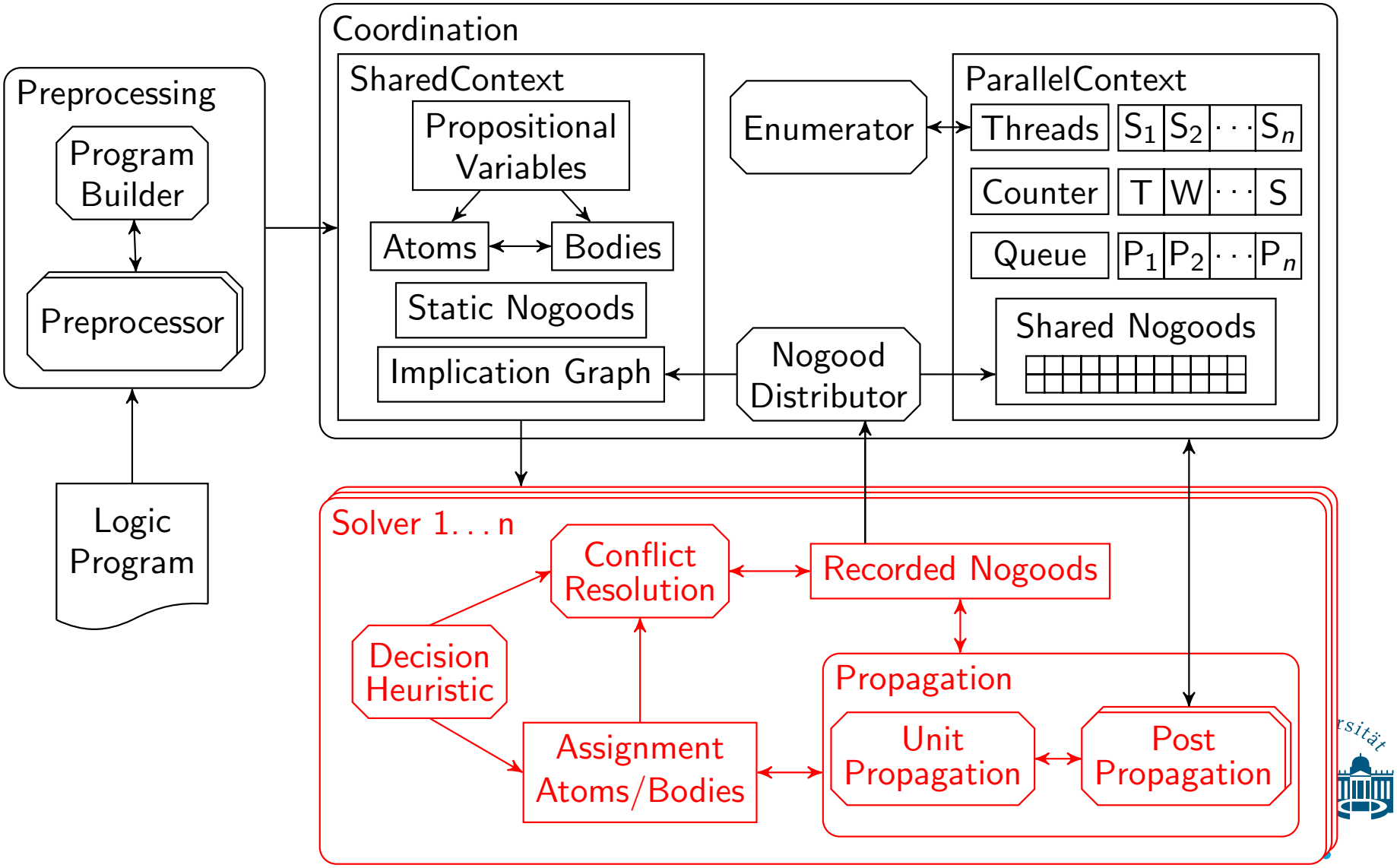
SharedContext

- The *SharedContext* object is initialized by the main thread and shared among all participating threads
- Among others, the *SharedContext* object contains
  - the set of relevant Boolean variables together with type information (eg atom, body, aggregate, etc),
  - a symbol table, mapping (named) atoms from the program to internal variables,
  - the positive atom-body dependency graph, restricted to its strongly connected components,
  - the set of Boolean constraints, among them nogoods, cardinality and weight constraints, minimize constraints, and
  - an implication graph capturing inferences from binary and ternary nogoods

Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Solver

- Each thread contains one *Solver* object, implementing parallel CDCL-style search
- Each *Solver* object stores
  - local data, including assignment, watch lists, constraint database, etc
  - local strategies, regarding heuristics, restarts, constraint deletion, etc

  and uses the *NogoodDistributor* to share recorded nogoods
- Each *Solver* object maintains a list of post propagators that are consecutively processed after unit propagation

Solver

- Each thread contains one *Solver* object, implementing parallel CDCL-style search
- Each *Solver* object stores
  - local data, including assignment, watch lists, constraint database, etc
  - local strategies, regarding heuristics, restarts, constraint deletion, etc

  and uses the *NogoodDistributor* to share recorded nogoods
- Each *Solver* object maintains a list of post propagators that are consecutively processed after unit propagation

Post Propagation

- Post propagators are assigned different priorities and are called in priority order
- Existing post propagators include
  - Unfounded-set checking
  - Failed-literal detection
  - Theory propagation (in *clingcon*)
- Parallelism is also handled by means of post propagators:
  - a high-priority post propagator for message handling and
  - a low-priority post propagator for integrating information

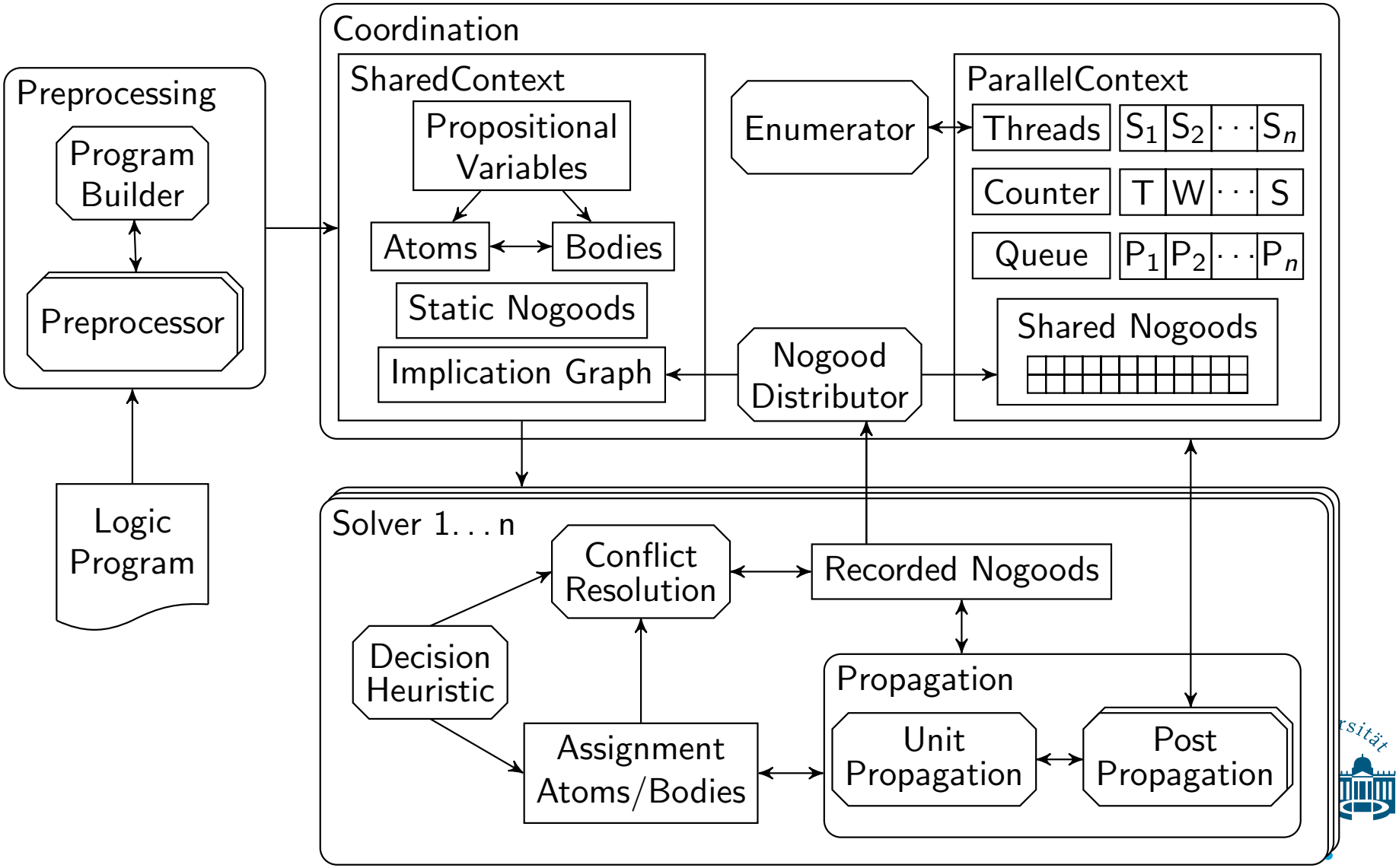
Post Propagation

- Post propagators are assigned different priorities and are called in priority order
- Existing post propagators include
  - Unfounded-set checking
  - Failed-literal detection
  - Theory propagation (in *clingcon*)
- Parallelism is also handled by means of post propagators:
  - a high-priority post propagator for message handling and
  - a low-priority post propagator for integrating information

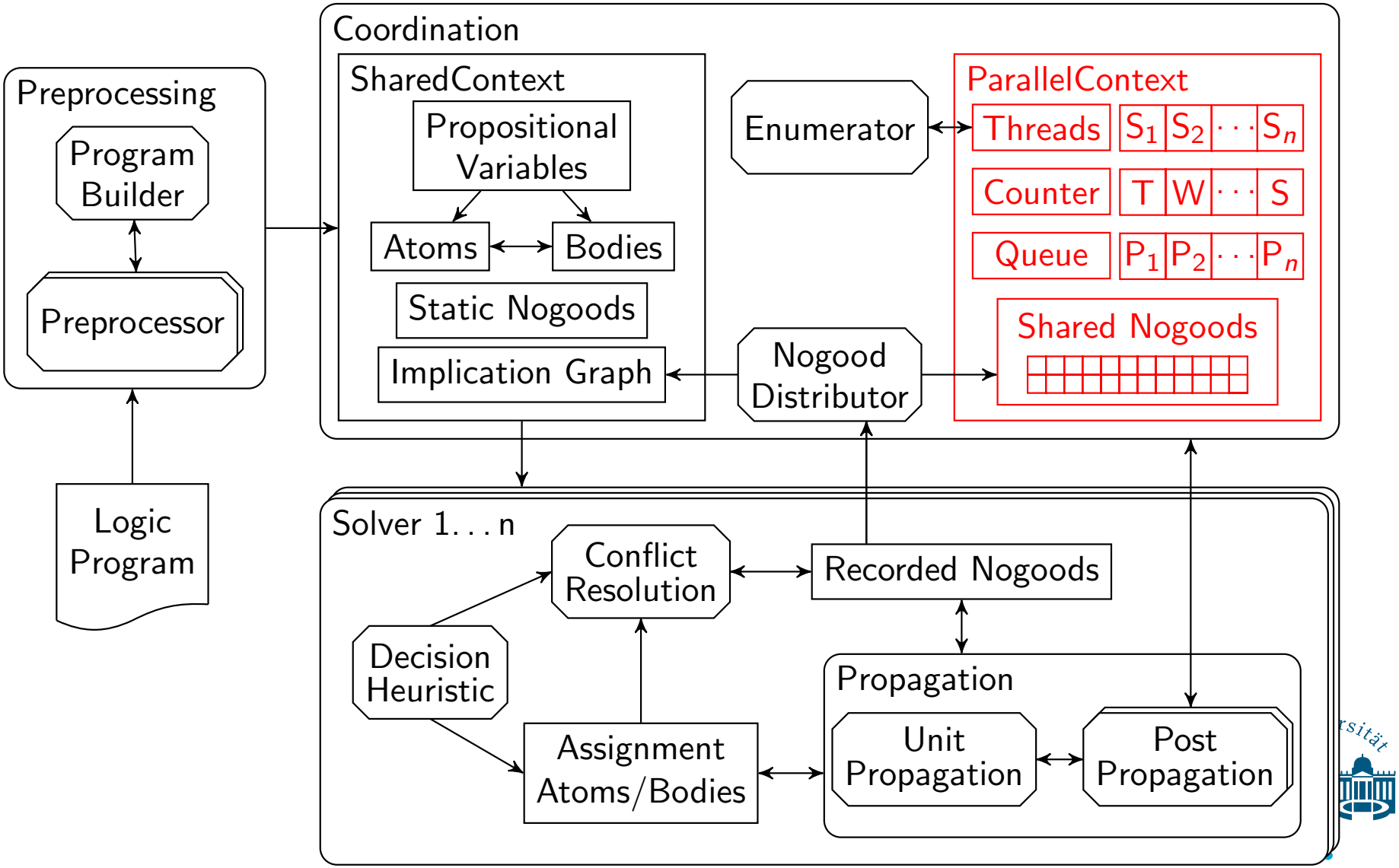
Post Propagation

- Post propagators are assigned different priorities and are called in priority order
- Existing post propagators include
  - Unfounded-set checking
  - Failed-literal detection
  - Theory propagation (in *clingcon*)
- Parallelism is also handled by means of post propagators:
  - a high-priority post propagator for message handling and
  - a low-priority post propagator for integrating information

Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



ParallelContext

- For controlling parallel search, the *ParallelSolve* object maintains a set of atomic message flags:
  - terminate signals the end of a computation,
  - interrupt forces outside termination (eg Ctrl+C),
  - sync indicates that all threads shall synchronize, and
  - split is set during splitting-based search whenever at least one thread needs work
- These flags are used to implement *clasp*'s two major search strategies:
  - splitting-based search via distribution of guiding paths and dynamic load balancing by means of a split-request and -response protocol, and
  - competition-based search via freely configurable solver portfolios
- Solver portfolios can also be used in splitting-based search!

ParallelContext

- For controlling parallel search, the *ParallelSolve* object maintains a set of atomic message flags:
  - terminate signals the end of a computation,
  - interrupt forces outside termination (eg Ctrl+C),
  - sync indicates that all threads shall synchronize, and
  - split is set during splitting-based search whenever at least one thread needs work
- These flags are used to implement *clasp*'s two major search strategies:
  - splitting-based search via distribution of guiding paths and dynamic load balancing by means of a split-request and -response protocol, and
  - competition-based search via freely configurable solver portfolios
- Solver portfolios can also be used in splitting-based search!

ParallelContext

- For controlling parallel search, the *ParallelSolve* object maintains a set of atomic message flags:
  - terminate signals the end of a computation,
  - interrupt forces outside termination (eg Ctrl+C),
  - sync indicates that all threads shall synchronize, and
  - split is set during splitting-based search whenever at least one thread needs work
- These flags are used to implement *clasp*'s two major search strategies:
  - splitting-based search via distribution of guiding paths and dynamic load balancing by means of a split-request and -response protocol, and
  - competition-based search via freely configurable solver portfolios
- Solver portfolios can also be used in splitting-based search!

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Communication architecture

- Thread Coordination
  - relies on message passing, efficiently implemented by lock-free atomic integers.
- Nogood Exchange
  - is controlled by separate distribution and integration components
- Complex Reasoning Modes
  - regular and projected model enumeration
  - intersection and union of models
  - uniform and hierarchical (multi-criteria) optimization
  - as well as combinations thereof

- See paper for details!

Communication architecture

- Thread Coordination
  - relies on message passing, efficiently implemented by lock-free atomic integers.
- Nogood Exchange
  - is controlled by separate distribution and integration components
- Complex Reasoning Modes
  - regular and projected model enumeration
  - intersection and union of models
  - uniform and hierarchical (multi-criteria) optimization
  - as well as combinations thereof

- See paper for details!

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Implementation

- Clear distinction between three types of data representations
  - read-only
  - shared
  - thread-local
- For instance, constraints are typically separated into
  - a thread-local part usually containing search-specific and thus dynamic data and
  - a (possibly shared) read-only part typically comprises static data not being subject to change
- See paper for details!

Implementation

- Clear distinction between three types of data representations
  - read-only
  - shared
  - thread-local
- For instance, constraints are typically separated into
  - a thread-local part usually containing search-specific and thus dynamic data and
  - a (possibly shared) read-only part typically comprises static data not being subject to change

- See paper for details!

Implementation

- Clear distinction between three types of data representations
  - read-only
  - shared
  - thread-local
- For instance, constraints are typically separated into
  - a thread-local part usually containing search-specific and thus dynamic data and
  - a (possibly shared) read-only part typically comprises static data not being subject to change
- See paper for details!

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

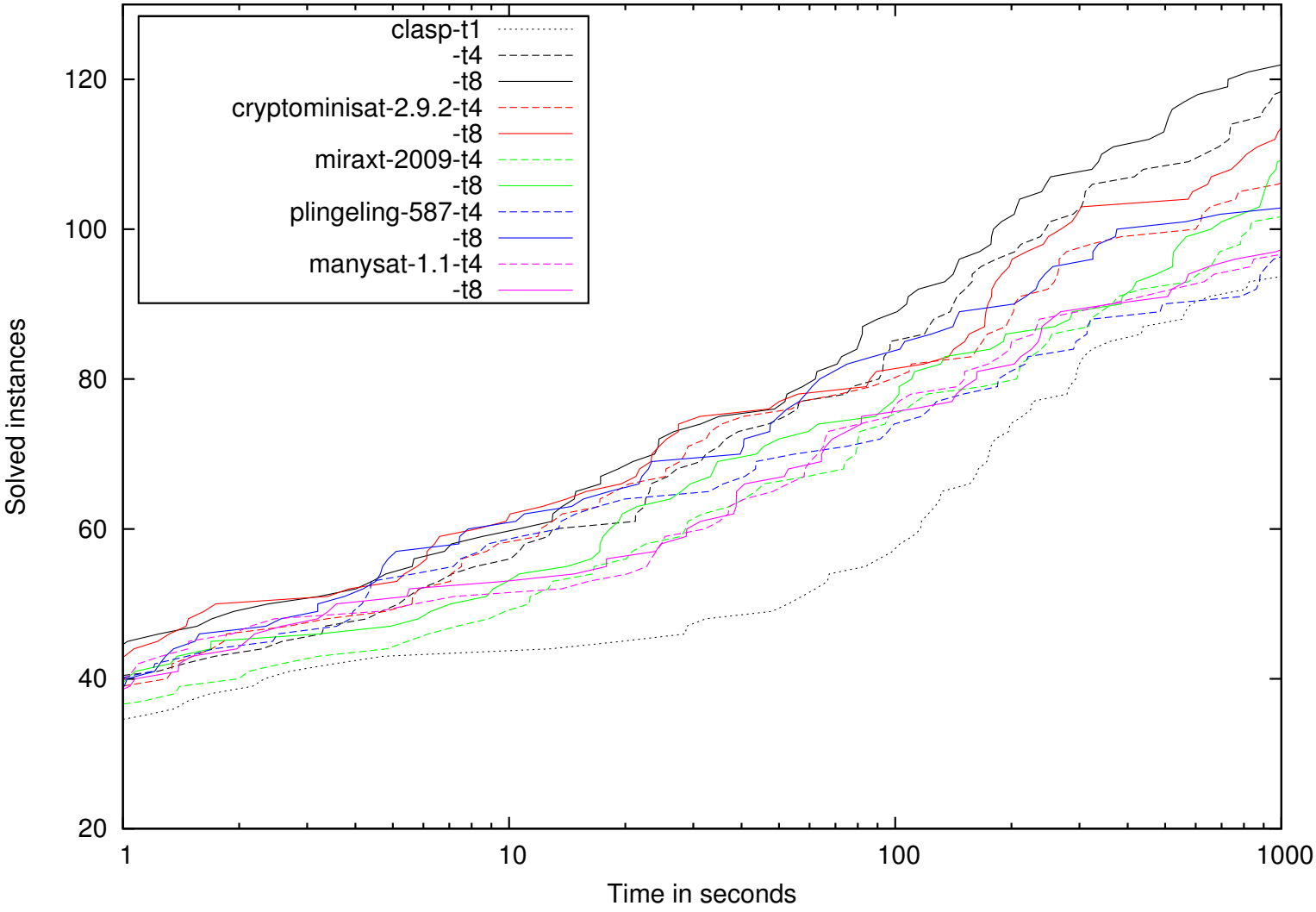
5 Summary

clasp in context

- Compare *clasp* (2.0.5) to the multi-threaded SAT solvers
  - *cryptominisat* (2.9.2)
  - *manysat* (1.1)
  - *miraxt* (2009)
  - *plingeling* (587f)

  all run with four and eight threads in their default settings
- 160/300 benchmarks from crafted category at SAT'11
  - all solvable by *ppfolio* in 1000 seconds
  - crafted SAT benchmarks are closest to ASP benchmarks

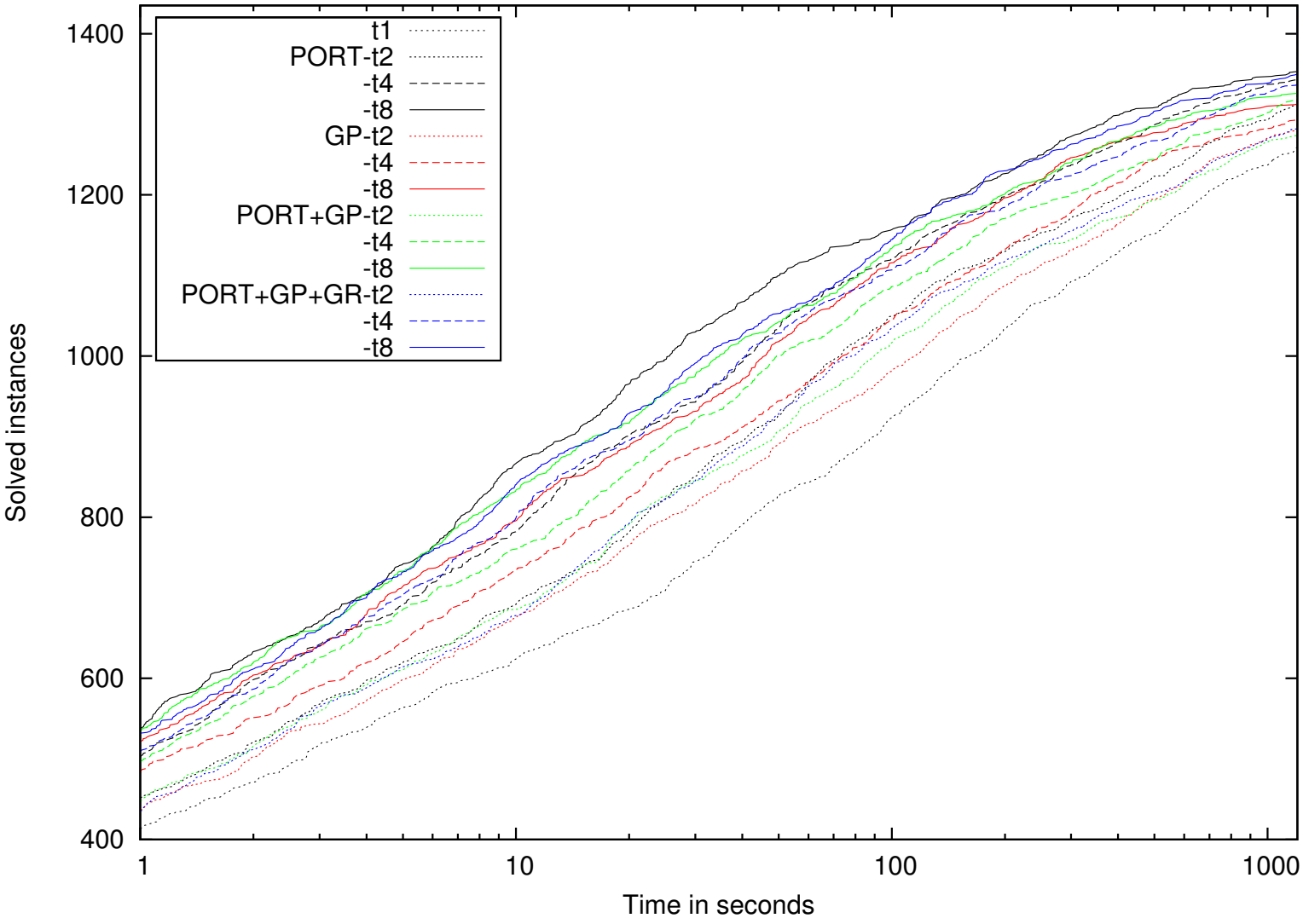
clasp in context



Impact of parallel search strategies

- Compare parallel search strategies
  - portfolio of competing threads (PORT)
  - search space splitting via guiding paths (GP)
  - splitting-based search with a portfolio of different configurations (PORT+GP)
  - previous setting plus global restarts (PORT+GP+GR)
- 1435 benchmark instances from ASP and SAT competitions

Impact of parallel search strategies

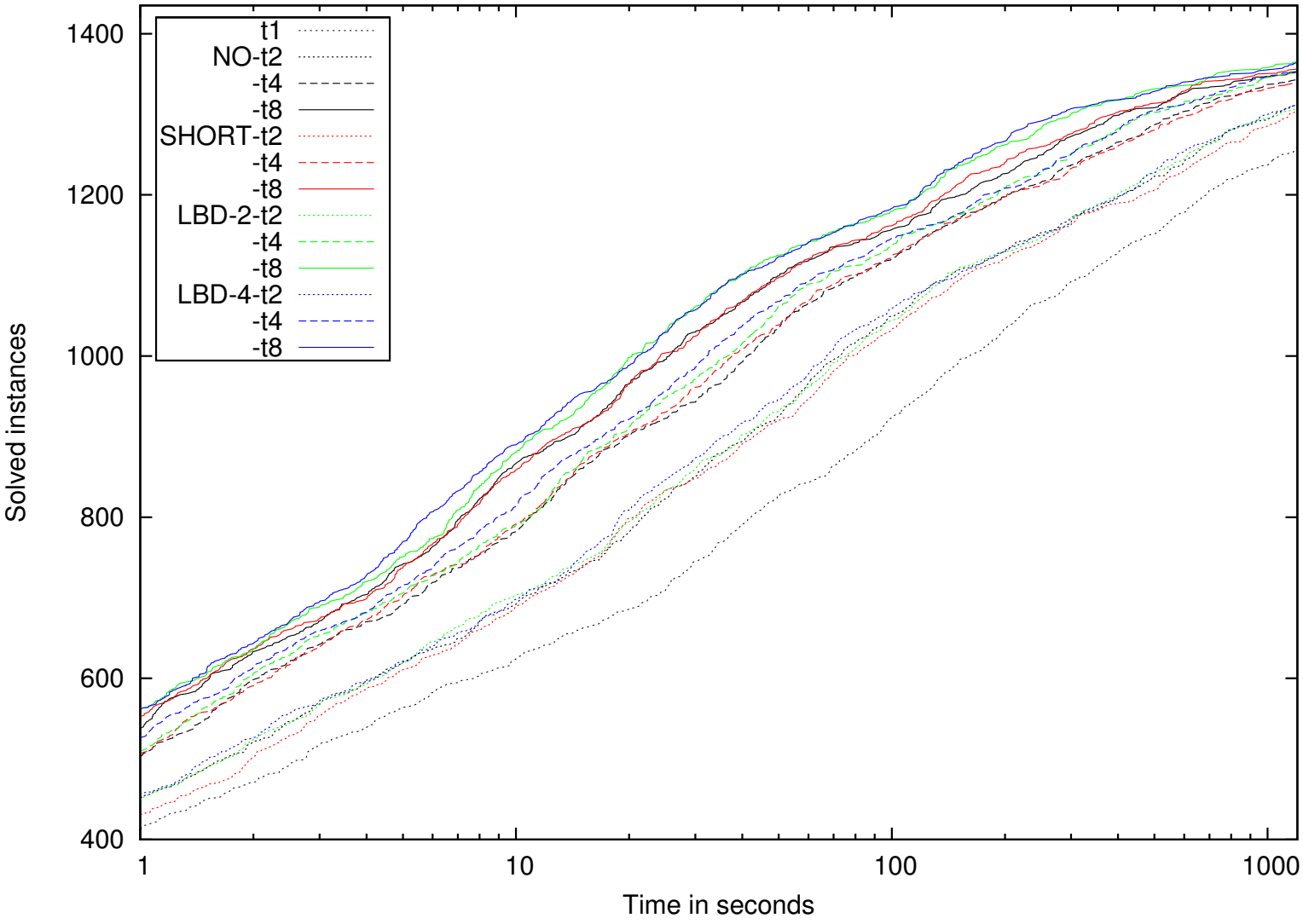


Impact of nogood exchange policies

- Compare nogood exchange policies of *clasp* (PORT)
  - short nogoods are shared "silently" (NO)
  - short nogoods are shared and communicated (SHORT)
  - nogoods with LBD 2 are shared and communicated (LBD-2)
  - nogoods with LBD 4 are shared and communicated (LBD-4)

  (LBD stands for Literal Block Distance)
- 1435 benchmark instances from ASP and SAT competitions

Impact of nogood exchange policies



clasp 2.1

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

clasp 2.1

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

clasp 2.1

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

clasp 2.1

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

clasp 2.1

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Outline

1 Introduction

2 Answer Set Solving

3 Multi-threaded Answer Set Solving
- Component architecture
- Communication architecture
- Implementation

4 Experiments

5 Summary

Summary

clasp 2 is a CDCL-based solver

- supporting parallelization via multi-threading
  - enhancing robustness (via up to 64 threads)
- featuring
  - different parallel search strategies
  - nogood exchange policies
  - various input formats
    - *smodels* (ASP)
    - *dimacs* (SAT and MaxSAT)
    - *opb* and *wbo* (PB)

Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`

Summary

clasp 2 is a CDCL-based solver

- supporting parallelization via multi-threading
  - enhancing robustness (via up to 64 threads)
- featuring
  - different parallel search strategies
  - nogood exchange policies
  - various input formats
    - *smodels* (ASP)
    - *dimacs* (SAT and MaxSAT)
    - *opb* and *wbo* (PB)

Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`

Summary

clasp 2 is a CDCL-based solver

- supporting parallelization via multi-threading
  - enhancing robustness (via up to 64 threads)
- featuring
  - different parallel search strategies
  - nogood exchange policies
  - various input formats
    - *smodels* (ASP)
    - *dimacs* (SAT and MaxSAT)
    - *opb* and *wbo* (PB)

Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`

Summary

clasp 2 is a CDCL-based solver

- supporting parallelization via multi-threading
  - enhancing robustness (via up to 64 threads)
- featuring
  - different parallel search strategies
  - nogood exchange policies
  - various input formats
    - *smodels* (ASP)
    - *dimacs* (SAT and MaxSAT)
    - *opb* and *wbo* (PB)

Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`