Automatic Cost Analysis for Imperative BSP programs

Arvid Jakobsson

HLPP 2017, Valladolid, Spain

HUAWEI
Huawei Research France

University of Orleans

Bulk Synchronous Parallel (1)

- **Bulk Synchronous Parallel (BSP)**: simple but powerful model for "semi-synchronous" data-parallelism
- BSP computer: (1) fixed number p of processor-memory pairs, (2) pair-wise communication network, (3) global synchronization unit
- BSP computation: Sequence of *super-steps*
- Super-step is composed of:
  1. Local computation by each process,
  2. Communication between processes,
  3. Synchronization barrier.

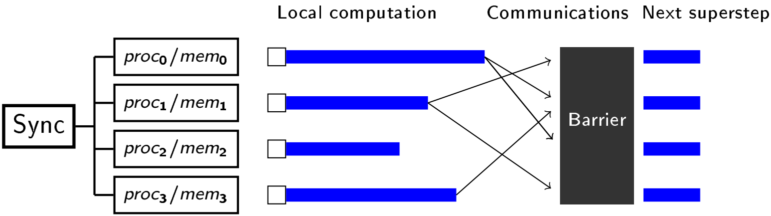


Figure: A BSP computer executing a superstep

Bulk Synchronous Parallel (2)

- Many implementations: BSPlib, BSML, BSP-Python, most linear algebra packages...
- DSLs such as Pregel and MapReduce are BSP-like
- Benefits of BSP:
  - Deadlock and data race free
  - Simplifies algorithm design
  - Simple but realistic cost model
    $\Rightarrow$ Scalable and Predictable performance in a portable and possibly immortal way

Bulk Synchronous Parallel (2)

- Many implementations: BSPlib, BSML, BSP-Python, most linear algebra packages...
- DSLs such as Pregel and MapReduce are BSP-like
- Benefits of BSP:
  - Deadlock and data race free
  - Simplifies algorithm design
  - Simple but realistic cost model
    $\Rightarrow$ Scalable and Predictable performance in a portable and possibly immortal way
  - Goal of this work:
    $\Rightarrow$ Automatic, Scalable and Predictable performance

BSP Cost model: BSP computer characterization

- BSP cost model: parallel architecture characterized by 4 parameters:
  - p: number processing units
  - r: cost of local computation
  - g: cost of communicating a 1-relation
  - l: cost of one barrier synchronization

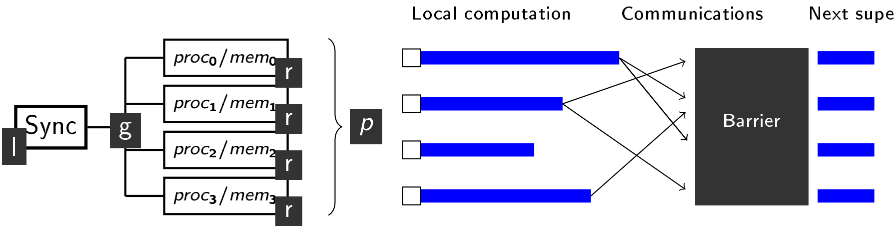


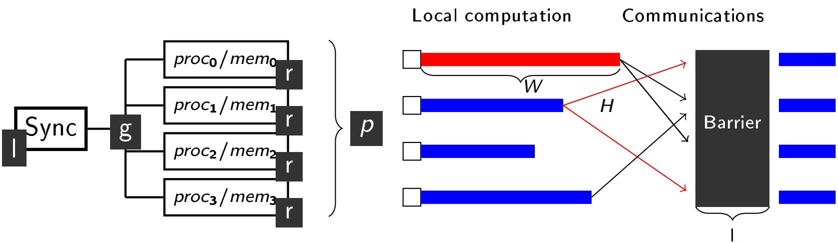
Figure: BSP computer characterization

BSP Cost model: Cost of BSP algorithms

- Cost of BSP computation: sum of cost of each super-step
- Cost of super-step:

$$Wr + Hg + l$$

  - W: Longest local computation
  - H: Maximum number of words received/sent by any process



- Cost of BSP algorithm: worst-case BSP computation cost expressed in algorithm's input variables

BSP Cost model: Example

- Example: Scan-algorithm for computing parallel prefix of the p-vector `x` (one component per process)

$s_{\text{scan}} =$

```
i := 1
while i < nprocs do
    if pid ≥ i then
        get(pid − i, x, x_in)
    end
    sync
    if pid ≥ i then
        {1*r } x := x + x_in
    end
    i := i * 2
end
```

- BSPlib-like, SPMD language
- Buffered DRMA communication: get

BSP Cost model: Example

- Example: Scan-algorithm for computing parallel prefix of the p-vector `x` (one component per process)

$s_{\text{scan}} =$

```
1:  i := 1
2:  while i < nprocs do
3:      if pid ≥ i then
4:          get(pid − i, x, x_in)
5:      end
6:      sync
7:      if pid ≥ i then
8:          {1* r } x := x + x_in
9:      end
10:     i := i * 2
11: end
```

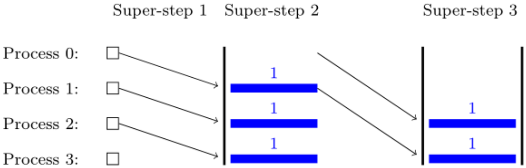
- Instruction's local computation costs is determined uniquely by annotation: $\{e * r\}\ i$
- In this example: only assignment on line 8 is counted

BSP Cost model: Example

- Example: Scan-algorithm for computing parallel prefix of the p-vector `x` (one component per process)

$s_{\text{scan}} =$

```
i := 1
while i < nprocs do
    if pid ≥ i then
        get(pid − i, x, x_in)
    end
    sync
    if pid ≥ i then
        {1*r } x := x + x_in
    end
    i := i * 2
end
```



- Execution with $p = 4$
- Cost of this execution:

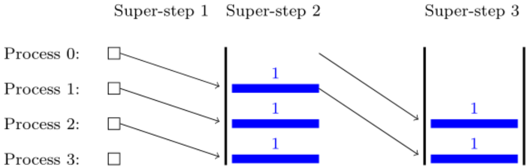
$$(0r + 1g + 1l) + (1r + 1g + 1l) + (1r + 0g + 1l) = 2r + 2g + 3l$$

BSP Cost model: Example

- Example: Scan-algorithm for computing parallel prefix of the p-vector `x` (one component per process)

$s_{\text{scan}} =$

```
i := 1
while i < nprocs do
    if pid ≥ i then
        get(pid − i, x, x_in)
    end
    sync
    if pid ≥ i then
        {1*r } x := x + x_in
    end
    i := i * 2
end
```



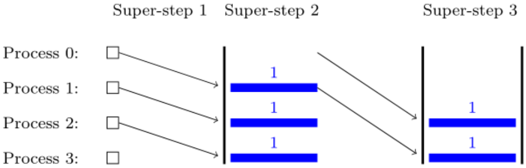
- Cost of Scan-algorithm: $(\log p)r + (\log p)g + (\log p + 1)l$
- Predictable performance on any BSP computer

BSP Cost model: Example

- Example: Scan-algorithm for computing parallel prefix of the p-vector `x` (one component per process)

$s_{\text{scan}} =$

```
i := 1
while i < nprocs do
    if pid ≥ i then
        get(pid − i, x, x_in)
    end
    sync
    if pid ≥ i then
        {1*r } x := x + x_in
    end
    i := i * 2
end
```



- Goal: Automatically obtain cost of imperative BSP programs

Automatic Cost Analysis: motivation

Downsides to manual cost analysis

- Tedious and error-prone
- Not always feasible / desirable:
  - On the fly scheduling,
  - Untrusted code,
  - Prototyping, etc.

Idea: re-use existing cost analysis for sequential program

- Automatic cost analyses exist for sequential programs:
  - Annotations give cost of individual instructions
  - Annotations can include cost for different resources: Example:

    $$\{1 * i + 1 * f\}\ x := (1 + 2)/2.0$$

    Interpretation: *Cost is one integer and one floating-point operation.*
  - Give the worst-case execution cost of input program
  - Conservative: results are upper-bounds

Idea: re-use existing cost analysis for sequential program

- Automatic cost analyses exist for sequential programs:
  - Annotations give cost of individual instructions
  - Annotations can include cost for different resources: Example:

    $$\{1 * i + 1 * f\}\ x := (1 + 2)/2.0$$

    Interpretation: *Cost is one integer and one floating-point operation.*
  - Give the worst-case execution cost of input program
  - Conservative: results are upper-bounds

- Idea: transform the BSPlib program to program whose *sequential costs upper-bounds the parallel cost* and use sequential cost analysis

Imperative BSP program $\longrightarrow$ Transformation $\longrightarrow$ Sequential Cost Analysis $\longrightarrow$ BSP Cost

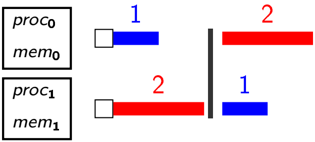
Challenge 1: Unrestricted divergence of control-flow

- Challenge 1: The longest time path of the BSP computation might not correspond to *any* sequential path

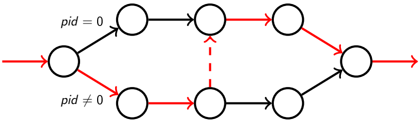
Challenge 1: Unrestricted divergence of control-flow

- Challenge 1: The longest time path of the BSP computation might not correspond to *any* sequential path

if $pid = 0$ **then**
$\{1 * r\}$ $Work_1(); sync; \{2 * r\}$ $Work_2();$
else
$\{2 * r\}$ $Work_2(); sync; \{1 * r\}$ $Work_1();$
end if



(a) BSP Execution



(b) Control Flow Graph

Solution 1: Requiring textually aligned programs

- Solution 1: Requiring *textually aligned barriers*
- When processes call `sync` in textually aligned program:
  - same call-site
  - same loop-iteration

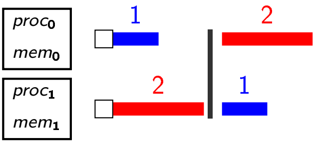
Solution 1: Requiring textually aligned programs

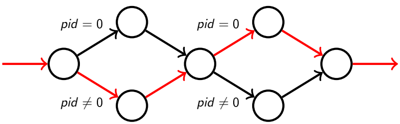
- Solution 1: Requiring *textually aligned barriers*
- When processes call `sync` in textually aligned program:
  - same call-site
  - same loop-iteration
- Verified using static analysis (see [JDB$^+$17])
- BSPlib (and MPI) programs are mostly written in this way [JDB$^+$17] ([ZD07])

Solution 1: Requiring textually aligned programs

- Solution 1: Requiring *textually aligned barriers*
- When processes call `sync` in textually aligned program:
  - same call-site
  - same loop-iteration

if $pid = 0$ **then** $\{1 * r\}$ $Work_1()$; **else** $\{2 * r\}$ $Work_2()$; **end if**
sync;
if $pid = 0$ **then** $\{2 * r\}$ $Work_2()$; **else** $\{1 * r\}$ $Work_1()$; **end if**



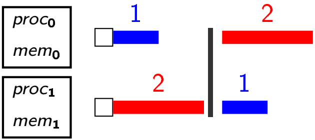


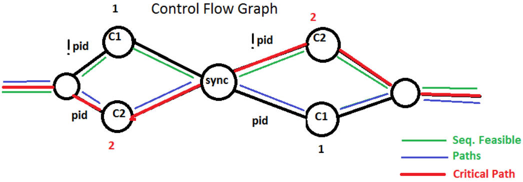
Challenge 2: Sequential vs. Parallel Longest Time Path

- Challenge 2: Longest time path of BSP computation might not be *feasible*

if $pid = 0$ **then** $\{1 * r\}$ $Work_1()$; **else** $\{2 * r\}$ $Work_2()$; **end if**
sync;
if $pid = 0$ **then** $\{2 * r\}$ $Work_2()$; **else** $\{1 * r\}$ $Work_1()$; **end if**

- Here: *pid* would evaluate to different values in 1st and 2nd guard





Solution 2: Non-deterministic scheduling

- Solution 2: Non-deterministic scheduling
- Instrument program to non-deterministically change state to any process before each super-step

Solution 2: Non-deterministic scheduling

- Solution 2: Non-deterministic scheduling
- Instrument program to non-deterministically change state to any process before each super-step

$\underline{pid := \texttt{any};}$
if $pid = 0$ **then** $\{1 * r\}$ $Work_1()$; **else** $\{2 * r\}$ $Work_2()$; **end if**
sync; $\underline{pid := \texttt{any};}$
if $pid = 0$ **then** $\{2 * r\}$ $Work_2()$; **else** $\{1 * r\}$ $Work_1()$; **end if**

Solution 2: Non-deterministic scheduling

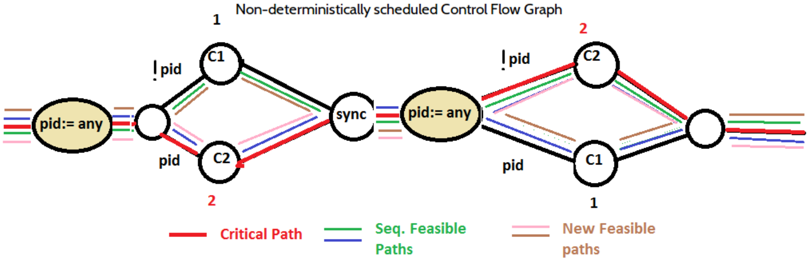
- Solution 2: Non-deterministic scheduling
- Instrument program to non-deterministically change state to any process before each super-step

$\underline{pid := \texttt{any};}$
if $pid = 0$ **then** $\{1 * r\}$ $Work_1()$; **else** $\{2 * r\}$ $Work_2()$; **end if**
sync; $\underline{pid := \texttt{any};}$
if $pid = 0$ **then** $\{2 * r\}$ $Work_2()$; **else** $\{1 * r\}$ $Work_1()$; **end if**

- Updates all variables with different values in two processes (e.g. *pid*)
- These variables are statically over-approximated

Solution 2: Non-deterministic scheduling

- Solution 2: Non-deterministic scheduling
- Instrument program to non-deterministically change state to any process before each super-step



- Longest time path feasible in instrumented program

Challenge 3: Communication distribution

- Challenge 3: Analyzing communication distribution

$s_{\text{scan}} =$

```
i := 1
while i < nprocs do
    if pid ≥ i then
        get(pid − i, x, x_in)
    end
    sync
    if pid ≥ i then
        {1*r } x := x + x_in
    end
    i := i * 2
end
```

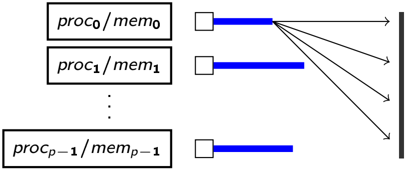
- BSP communication cost: ?
- Depends on how target expression *pid* − *i* evaluates in all processes

Challenge 3: Communication distribution

- Example:

$$get(pid - i, x, x_{in})$$

- Target expression $pid - i$ has same value in all processes, e.g. $i = pid$ and so $pid - i = 0$.



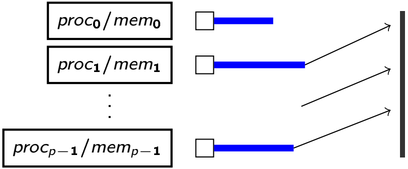
- BSP communication cost: pg
- Potential annotation: $\{p * g\}$

Challenge 3: Communication distribution

- Example:

$$get(pid - i, x, x_{in})$$

- Target expression $pid - i$ has distinct value on all processes, e.g. $i = 1$



- BSP communication cost: $1g$
- Potential annotation: $\{1 * g\}$

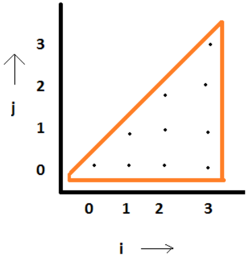
Challenge 3: Communication distribution

- Conclusion: must conservatively assume unbalanced communication or analyze the target expression more precisely

Solution 3: Polyhedral model & Communicating section

- Precise reasoning on communication by polyhedral model
- Polyhedral model: executions of statement S as set of points

```
for i ∈ [0..3] do
    for j ∈ [0..i] do
        S : ...
    end for
end for
```



$$\mathcal{D} = \{[i,j] \in \mathbb{Z}^2 \mid 0 \leq i \leq 3 \wedge \\ 0 \leq j \leq i\}$$

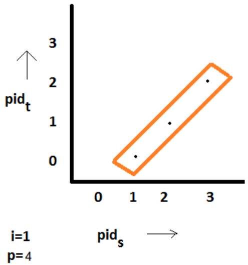
- Bounds on loop-iterators i must be affine combinations of outer iterators, e.g. $i \leq aj + bk + \cdots + c$.

Solution 3: Polyhedral model & Communicating section

- Requirements:
  - textually aligned communicating section
  - replicated parameters
  - affine target expression
- Add two axes to polyhedra:
  - $\mathtt{pid}_s \in [0..p)$ (source process) and
  - $\mathtt{pid}_t = \textit{target expression}$ (target process)
- Obtain the $\boxed{\text{interaction set}}$

Solution 3: Polyhedral model & Communicating section

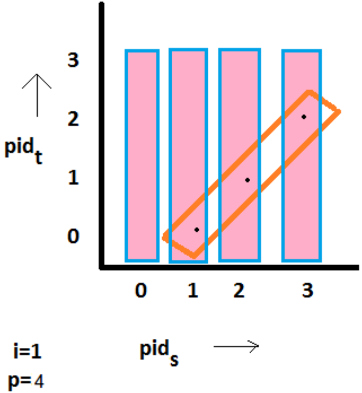
if $pid \geq i$ **then**
    $get(pid - i, x, x_{in})$
end if

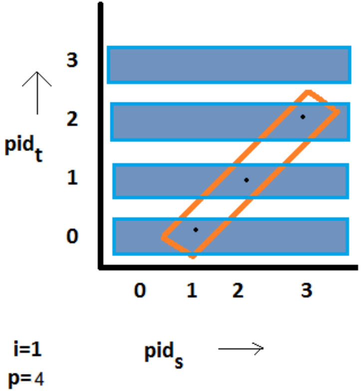


$$\mathcal{D} = \{[\mathtt{pid}_s, \mathtt{pid}_t] \in \mathbb{Z}^2 \mid 0 \leq \mathtt{pid}_s < p \wedge \\ \mathtt{pid}_t = \mathtt{pid}_s - i \wedge \\ \mathtt{pid}_s \geq i\}$$

Solution 3: From interaction set to H-relation

- How to go from $\boxed{\text{interaction set}}$ to H-relation?
- Partition set twice: outbound & inbound requests per `pid`:





- H-relation = largest part. For this example, H-relation = 1
- Efficiently computed by polyhedral libraries

Solution 3: Insert bound in program

- Transformation annotates entry of communicating section with H-relation

```
pid := any;
i := 1
while i < nprocs do
    {1*g}
    if pid ≥ i then
        skip
    end
    {1 l} skip
    pid := any; x := any; x_in := any;
    if i < nprocs then
        {1*w} x := x + x_in
    end
    i := i × 2
end
{1*l} skip
```

- H-relation annotation
- Communicating section
- Non-deterministic scheduling

Send program to sequential cost analysis

- Last step: send the instrumented program with communication bounds to the sequential cost analysis
- Result: upper-bound on BSP cost, parametric in input-variables

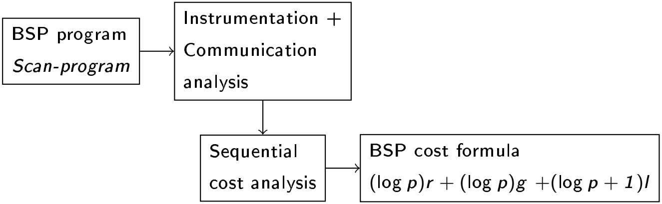


Figure: Analysis pipeline

Implementation

- Prototype in Haskell: 3000 lines (includes sequential cost analysis)
  - APRON (Numerical abstract domain library) [JM09] for abstract interpretation
  - PUBS (Practical upper-bounds solver) [AAGP11] for cost equation solving
  - Simple pattern matching for extracting polyhedra
  - `isl` [Ver10] (Integer Set Library) for operations on polyhedra

Evaluation & Limitations

Program	Result		
Scan $N = p$	$(\log p)r$	$+(\log p)g$	$+(\log p + 1)l$
Scan $N > p$	$(2N/p + p - 2)r$	$+(p-1)g$	$+2l$
Compress	$(3N/p + p - 2)r$	$+Ng$	$+3l$
Broadcast $N > p$ (1)		$(p-1)Ng$	$+2l$
Broadcast $N > p$ (2)		$(\log p)Ng$	$+(\log p + 1)l$
Broadcast $N > p$ (3)		$2(p-1)N/pg$	$+3l$
Fold	$(N/p + p - 2)r$	$+pg$	$+2l$

- Evaluated method on BSP communication patterns from text-book
- Precise cost for all but Compress: data-dependent communication pattern
- When control-flow and communication pattern is not data-dependent: works well

Future work & Conclusion

Contributions

- Method and prototype for analyzing cost of imperative BSP program
- Evaluation on small programs with promising results

Future work

- Implementation and experiment on larger programs
- Prove correctness of transformation
- Handling data-dependent control flow
- Consider other measures on BSP costs: best-case, average-case, etc.

References

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla, *Closed-form upper bounds in static cost analysis*, Journal of Automated Reasoning **46** (2011), no. 2, 161–203.

Arvid Jakobsson, Frederic Dabrowski, Wadoud Bousdira, Frédéric Loulergue, and Gaetan Hains, *Replicated Synchronization for Imperative BSP Programs*, International Conference on Computational Science (ICCS) (Zürich, Switzerland), Procedia Computer Science, Elsevier., 2017.

Bertrand Jeannet and Antoine Miné, *Apron: A library of numerical abstract domains for static analysis*, International Conference on Computer Aided Verification, Springer, 2009, pp. 661–667.

Sven Verdoolaege, *Isl: An integer set library for the polyhedral model*, International Congress on Mathematical Software, Springer, 2010, pp. 299–302.

Yuan Zhang and Evelyn Duesterwald, *Barrier Matching for Programs with Textually Unaligned Barriers*, Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA), PPoPP '07, ACM, 2007, pp. 194–204.