Partial Globalization of Partitioned Address Spaces for Zero-copy Communication with Shared Memory

Fangzhou Jiao, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, Andrew Lumsdaine
Indiana University

HiPC 2011
Motivation

• Increasing popularity and availability of many-cores
• Abundance of legacy MPI code
• Simplifying programming model
  • single model, instead of hybrid
• Leveraging shared memory fully for performance
• Proving that shared memory could be used as an optimization for communication
Partitioned Address Space Programming on Shared Memory

- Avoids having to worry about race conditions
- Encourages programmers to think about locality
- Could make it easier to reason about program correctness
  - if done at the right level of abstraction

Needs special handling to compete in performance with threaded shared memory programs
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
- alternate between computation and communication
- communication optimization breaks the structure
Declarative Approach

• Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication

• alternate between computation and communication

• communication optimization breaks the structure
Declarative Approach

• Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication

• alternate between computation and communication

• communication optimization breaks the structure

• Extend to non BSP-style applications
Kanor for Clusters

@communicate { b@recv_rank <<= a@send_rank }


Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Kanor for Clusters

@communicate \{ b@recv\_rank <= a@send\_rank \}

e_0@e_1 <= op <= e_2@e_3 where e_4


Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

e_0@e_1 << op << e_2@e_3 where e_4

e_0@e_1 <= e_2@e_3 where e_4
Kanor for Clusters

@communicate \{ \texttt{b@recv\_rank} \ll= \texttt{a@send\_rank} \}

\[ e_0@e_1 \ll op \ll e_2@e_3 \text{ where } e_4 \]

\[ e_0@e_1 \ll= e_2@e_3 \text{ where } e_4 \]

\begin{itemize}
\item \texttt{A[j] @ i} \quad \text{storage location}
\item \texttt{receiver rank}
\item \texttt{B[i] @ j} \quad \text{reduction operator}
\item \text{data}
\item \texttt{where i in world, j in \{0...i\}, i \% 2 == 0} \quad \text{generator}
\item \text{filter}
\end{itemize}


Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Kanor for Clusters

@communicate { b@recv_rank <<= a@send_rank }

e_0 @ e_1 <<= op <<= e_2 @ e_3 where e_4

e_0 @ e_1 <<= e_2 @ e_3 where e_4

A[j] @ i <<= B[i] @ j where i in world, j in {0...i}, i % 2 == 0

storage location receiver rank reduction operator data sender rank generator generator filter

Source-level compiler (using ROSE)

standard C++ code


Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Design Principles

• Users must think in parallel (creativity)
  • but not be encumbered with optimizations that can be automated, or proving synchronization correctness

• Compiler focuses on what it can do (mechanics)
  • not creative tasks, such as determining data distributions, or creating new parallel algorithms

• Incremental deployment
  • not a new programming language
  • more of a coordination language (DSL)

• Formal semantics
  • provable correctness
Compiling to MPI

@communicate {x@1 <= x@0}

Node 0

Node 1

send x

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Compiling to MPI

@communicate \{x@1 \ll= x@0\}

Node 0

send x

Node 1

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Compiling to MPI

@communicate \{x\textsubscript{1} \ll= x\textsubscript{0}\}

Node 0

Node 1

send x

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Compiling to MPI

```
@communicate {x@1 <<= x@0}
```

Node 0

```
send x
```

Node 1

```
recv x
```

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Compiling to MPI

@communicate \{ x@1 \ll x@0 \}
Compiling to MPI

```
@communicate {x@1 <<= x@0}
```

3 copies
MPI Optimized for Shared Memory

@communicate \{x@1 \lll= x@0\}

Node 0

send x

Node 1

recv x

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
MPI Optimized for Shared Memory

@communicate \{x@1 \ll= x@0\}

Node 0

send x

Node 1

recv x

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
MPI Optimized for Shared Memory

@communicate {x@1 <= x@0}

Node 0

Node 1

send x

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
MPI Optimized for Shared Memory

@communicate {x@1 <<= x@0}

Node 0

send x

App

Node 1

recv x

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
MPI Optimized for Shared Memory

@communicate \{x@1 \ll= x@0\}

Node 0

\textbf{send x}

Node 1

\textbf{recv x}

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
MPI Optimized for Shared Memory

@communicate \{x@1 \ll= x@0\}

Node 0

Node 1

send x
recv x

App

MPI

2 copies
Optimizing for Shared Memory

@communicate {x@1 <<= x@1}

Node 0

send x

Node 1

recv x

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

@communicate {x@1 <<= x@1}

Node 0

send x

Node 1

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

```c
@communicate {x@1 <<= x@1}
```

Node 0

 send x

Node 1

 recv x

App

mpi
Optimizing for Shared Memory

@communicate \{x@1 \ll= x@1\}

Node 0
send x

Node 1
recv x

App

MPI

1 copy
(requires rendezvous or compiler intervention)

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

@communicate {x@0 <= x@1}

Node 0

send x

Node 1

recv x

App

MPI
Optimizing for Shared Memory

@communicate \{x@0 \ll= x@1\}

Node 0

Node 1

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

@communicate {x@0 <<= x@1}

Node 0
signal(sem_x)

x

Node 1
wait(sem_x)

App

MPI
Optimizing for Shared Memory

@communicate {x@0 <= x@1}

Node 0

signal(sem_x)

wait(sem_x)

App

MPI

Node 1

x

0 copy

(requires compiler intervention)

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

```c
@communicate {x@0 <<= x@1}
```

@communicate

```
Node 0
signal(sem_x)

Node 1
wait(sem_x)
```

Partial Globalization

0 copy
(requires compiler intervention)
Steps for Optimizing Communication with Shared Memory

- Identify globalization candidates
- Ensure correctness
  - insert appropriate synchronization
- Minimize contention
  - minimize synchronization points
  - minimize synchronization overheads
  - using a run-time trick
Globalization Candidates

• Contiguous chunks of memory
  • excluding strided array sections, for example
  • contiguous array sections OK (but not implemented)

• Large buffers
  • communication inside loops

• Small local reuse
Ensuring Correctness

@communicate {x[i] <= x[i+1],
              where i in Kanor::WORLD}

...  
consume(A);   // consume communicated data

...  
overwrite(A); // reuse A for local data

...  
consume(A);   // consume local data
Ensuring Correctness

```cpp
@communicate {x[i] <= x[i+1],
    where i in Kanor::WORLD}
...
consume(A);   // consume communicated data
...
overwrite(A); // reuse A for local data
...
consume(A);   // consume local data
```
Correctness Issues

@communicate\{x@i <= x@0\}, where \( i > 0 \)

\[
\begin{align*}
\ldots &= x \\
\ldots &= x \\
\ldots &= x \\
\ldots &= x \\
x &= \ldots \\
x &= \ldots \\
x &= \ldots \\
x &= \ldots \\
\end{align*}
\]
Observations

**Definition:** Locking Set: The set of CFG nodes that lie on a path from a node containing local write into a globalized variable to a node containing read of that value.
Observations

**Definition:** Locking Set: The set of CFG nodes that lie on a path from a node containing local write into a globalized variable to a node containing read of that value.

**Theorem:** If the locking set belongs to a critical section then the partitioned address space semantics are maintained.
Correctness: Examples

@communicate ...

x = ...

... = x

E

x = ...

... = x

Algorithm 1: Algorithm to compute the set of all nodes that lie on any path from s to t.

by doing a BFS starting at s. Similarly, lines 14–15 do a backward BFS (on reverse edges) starting at t, which visits any node that can reach t. Thus, any node added to P is on a path from s to t. On the other hand, if there is a path from s to t then any node on that path must be reachable from s, and t must be reachable from any such node. Thus, the algorithm will discover that node in the BFS from s as well as in the reverse BFS from t, adding it to P. Finally, the two BFS steps in the algorithm lead directly to the time complexity of $O(|E| + |V|)$.

In order to arrive at an algorithm to compute the locking set, we make several observations in the form of following lemmas.

Lemma 1. For a loop-carried dependence, carried by loop level $l$, all dependence carrying edges in the CFG lie at loop level $l$ or higher and any dependence carrying path must traverse the looping back-edge at level $l$.

Proof: The proof follows directly from the definition of loop-carried dependencies [9].

Lemma 2. For a loop-independent dependence between state-statements that are at the common level $l$, no dependence carrying path in the CFG goes through the looping back-edge at level $l$ or lower.

Proof: If the looping back-edge at level $l$ was involved in the dependence it would be a loop-carried dependence.

Lemma 3. Suppose that there is a loop-carried true dependence from a CFG node $w$ to a CFG node $r$ with dependence distance 1 due to a variable $x$, carried by a loop with the head node $h$. Suppose that $P_{u,v}$ denotes the set of nodes on all possible simple paths from $u$ to $v$. Then, the locking set for $x$ due to the dependence from $w$ to $r$ is given by $L_x = P_{w,h}[P_{h,r}]$.

Proof: For a loop-carried dependence with dependence distance 1, any dependence-carrying path goes through the looping back-edge exactly once. Thus, any such path must start from the write node, $w$, go through the looping back-edge, and end at the read node, $r$. Therefore, the locking set for $x$ is the set of all nodes on any simple path from $w$ to $r$.

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Correctness: Examples

![Diagram showing examples of control flow graphs (CFGs) illustrating the subtleties involved in computing locking sets.]

- For a loop-carried dependence with dependence distance 1, all dependence carrying edges in the CFG lie at loop level 1.
- Similarly, lines 14–15 do a loop-carried dependence from a CFG node.

Proof:

1. For a loop-carried dependence, carried by loop $P$, suppose that there is a loop-carried true dependence from a CFG node $u$ at level $l$.
2. The looping back-edge at level $l$ for a loop-carried dependence is given by $P[w,h]$.
3. For a loop-carried dependence with dependence distance 1 due to a variable $w$, looping back-edge exactly once.
4. Thus, any such path must visit $P[w,h]$ exactly once.
5. For a loop-carried dependence with dependence distance 1, any dependence-carrying path goes through the looping back-edge at level $l$.
6. For a loop-carried dependence with dependence distance 1, any dependence-carrying path must visit $P[w,h]$ at least once.
7. The proof follows directly from the definition of $P[w,h]$.
Correctness: Examples

@communicate ...

\[
x = \ldots
\]

\[
\ldots = x
\]

\[
x = \ldots
\]

\[
\ldots = x
\]

For a loop-independent dependence between state-
ments that are at the common level
ments that are at the common level
dence from a CFG node
dence from a CFG node
Proof:
Proof:
\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]

\[
\text{Proof:}
\]
Overall Algorithm

- Identify globalization candidates
- For each globalized variable
  - compute the *locking set*
  - divide the locking set into *connected components*, $C_i$
    - CFG edge into $C_i \Rightarrow$ insert `lock_acquire`
    - CFG edge out of $C_i \Rightarrow$ insert `lock_release`
Example of sub-optimal Behavior

@communicate ...

for ...

... = x

x = ...

@communicate ...

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Copy-on-conflict

```cpp
void acquire_or_copy (Buffer& a, Lock& lock) {
    if (Localized[a]) return NULL;
    Condition cond;
    enum {COPY_THRD, LOCK_THRD} notifier;
    a_cpy = new Buffer;

    Thread l_thrd =
        spawn(acquire_lock, lock, cond, &notifier);
    Thread c_thrd =
        spawn(buf_copy, a, a_cpy, cond, &notifier);
    wait(cond);

    if (notifier == LOCK_THRD) {
        c_thrd.kill();
        free(a_cpy);
    } else {
        l_thrd.kill();
        if (lock.held()) lock.release();
        delete a;
        a = a_cpy;
        Localized[a] = true;
    }
}
```
Copy-on-conflict

```c
void acquire_or_copy (Buffer& a, Lock& lock) {
    if (Localized[a]) return NULL;
    Condition cond;
    enum {COPY_THRD, LOCK_THRD} notifier;
    a_cpy = new Buffer;

    Thread l_thrd =
        spawn(acquire_lock, lock, cond, &notifier);
    Thread c_thrd =
        spawn(buf_copy, a, a_cpy, cond, &notifier);
    wait(cond);

    if (notifier == LOCK_THRD) {
        c_thrd.kill();
        free(a_cpy);
    } else {
        l_thrd.kill();
        if (lock.held()) lock.release();
        delete a;
        a = a_cpy;
        Localized[a] = true;
    }
}
```
**Experimental Evaluation**

<table>
<thead>
<tr>
<th>Op</th>
<th>Kanor</th>
<th>MPI</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>A[j]@i &lt;= A[i]@j where i,j in WORLD</td>
<td>MPI_Alltoall (...)</td>
<td>barrier();</td>
</tr>
<tr>
<td>b’cast</td>
<td>A@i &lt;= A@0 where i in WORLD</td>
<td>MPI_Bcast(A, ..., ..., 0, ...);</td>
<td>barrier();</td>
</tr>
<tr>
<td>shift</td>
<td>A@i &lt;= A@i+1</td>
<td>if (Rank == (numprocs - 1)) dest = 0; else dest = Rank + 1; MPI_Send(A, array_size, ...); MPI_Recv(A, array_size, ...);</td>
<td>barrier();</td>
</tr>
<tr>
<td>reduce</td>
<td>A@0 &lt;=op&lt;= A@i where i in WORLD</td>
<td>MPI_Reduce (...) // or specialized code for // tree-reduction of ‘op’</td>
<td>// loop for tree-reduction for (i ...) { A[i] = op(..); }</td>
</tr>
</tbody>
</table>

- 8-core AMD Opteron, Gentoo Linux, OpenMPI 1.4.3
- Case 1: No local writes
- Case 2: Local writes
  - 2a: lock successfully acquired
  - 2b: buffer copied locally before the lock could be acquired
- Case 3: Forced copying (overlapping live ranges)
Case 1

![Graph showing speedup over MPI for different input sizes and operations: alltoall, broadcast, shift, and reduction.](image-url)
Case 2a

Speedup over MPI

Input size

Input size

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Case 2b

Speedup over MPI

Input size

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Case 3

Speedup over MPI

Input size

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Concluding Remarks

• Parallel programming with partitioned address spaces has advantages

• Appropriate abstraction makes parallel programming more accessible to intermediate-level programmers
  • Kanor demonstrates the effectiveness of this approach

• Advantages of shared memory can be obtained through compiler optimizations
  • our compiler algorithms and experimental evaluation substantiate this claim
End
Optimizing for Shared Memory

@communicate \{x@0 \ll= x@1\}

Node 0

send x

Node 1

recv x

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

@communicate {x@0 \leq x@1}
Optimizing for Shared Memory

@communicate \{x@0 \leq x@1\}

Node 0

\texttt{x_1}  \quad \texttt{x_2}

\texttt{signal(sem_x)}

App

Node 1

\texttt{wait(sem_x)}
\texttt{copy x_1, x_2}
Optimizing for Shared Memory

@communicate \{x@0 \ll= x@1\}

Node 0

Node 1

signal(sem_x)

wait(sem_x)
copy x_1, x_2

App

MPI

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Optimizing for Shared Memory

@communicate \{x@0 \leq x@1\}
Optimizing for Shared Memory

@communicate \{x@0 \leq x@1\}

Node 0

<table>
<thead>
<tr>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal(sem_x)</td>
</tr>
</tbody>
</table>

Node 1

<table>
<thead>
<tr>
<th>x_2</th>
</tr>
</thead>
</table>
| wait(sem_x) 
| copy x_1, x_2 |

App

<table>
<thead>
<tr>
<th>x_1</th>
</tr>
</thead>
</table>

MPI

<table>
<thead>
<tr>
<th>x_2</th>
</tr>
</thead>
</table>

1 copy
(requires compiler intervention)

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011
Computing all Paths from s to t

Algorithm: PATHS

Input: Directed graph $G(V, E)$
   Start node $s$
   End node $t$

Output: Set $P$ of nodes that lie on any path from $s$ to $t$

$P \leftarrow \phi$

for each node $n$ in $G$ do
   $n$.color $\leftarrow$ “white”
   $Q \leftarrow [s]$
   while not $Q$.empty do
      $q \leftarrow Q$.extract
      for each edge $(q, v) \in E$ do
         if $v$.color $\neq$ “red” then
            $v$.color $\leftarrow$ “red”
            $Q$.add($v$)
      $Q \leftarrow [t]$
      while not $Q$.empty do
         $q \leftarrow Q$.extract
         for each edge $(v, q) \in E$ do
            if $v$.color $\neq$ “black” then
               if $v$.color = “red” then
                  $P \leftarrow P \cup \{v\}$
               $v$.color $\leftarrow$ “black”
               $Q$.add($v$)
      return $P$
Computing Locking Sets

**Algorithm**: \text{COMPUTE-LOCKING-SET} 

**Input**: CFG $G(V, E)$ of code region over which variable $x$ is globalized, with level-annotated nodes; 
- dependence levels, $l_x$, for dependencies involving $x$; 
- dep. distances, $d_x$, for dependencies involving $x$; 

**Output**: Locking set $L$

\begin{align*}
L &= \phi \\
\text{for each node pair } (w, r) \text{ with an entry in } l_x &\text{ do} \\
\quad \text{if } d_x(w, r) = 0 \text{ then} \\
\quad\quad \text{if } l_x(w, r) = 0 &\text{ then} \\
\quad\quad\quad L &\leftarrow L \cup \text{PATHS}(G, w, r) \\
\quad\quad\text{else} \\
\quad\quad\quad G'(V', E') &\leftarrow G \text{ without any looping back-edges at} \\
\quad\quad\quad \text{level } l_x(w, r) \text{ and lower} \\
\quad\quad\quad L &\leftarrow L \cup \text{PATHS}(G', w, r) \\
\quad\quad\text{else if } d_x(w, r) = 1 &\text{ then} \\
\quad\quad\quad h &\leftarrow \text{head node of loop at level } l_x(w, r) \\
\quad\quad\quad G'(V', E') &\leftarrow G \text{ restricted to levels } l_x(w, r) \text{ and higher} \\
\quad\quad\quad L &\leftarrow L \cup \text{PATHS}(G', w, h) \cup \text{PATHS}(G', h, r) \\
\quad\quad\text{else} \\
\quad\quad\quad G'(V', E') &\leftarrow G \text{ restricted to levels } l_x(w, r) \text{ and higher} \\
\quad\quad\quad L &\leftarrow L \cup \text{PATHS}(G', w, r) \\
\text{return } L
\end{align*}