北京大学高能效计算与应用中心
Center for Energy-efficient Computing and Applications

FPGA Routing 并行加速

2024年 并行与分布式导论

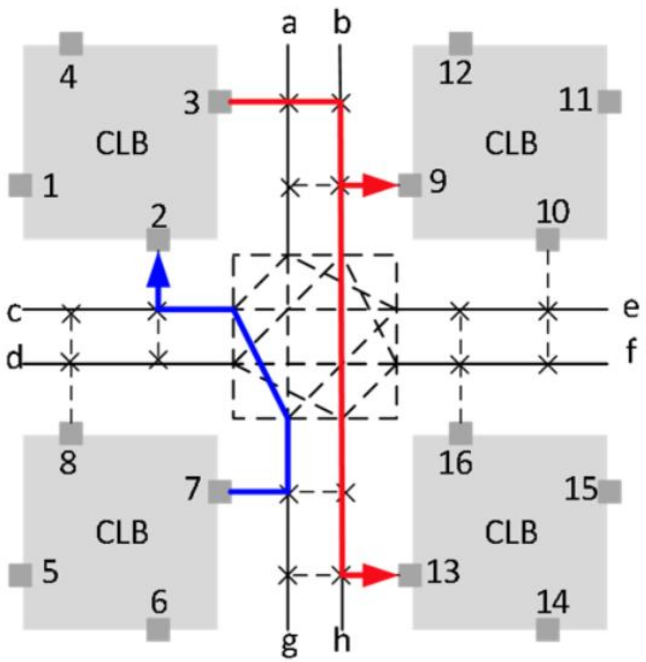
2024/05

OUTLINE

- 基本概念与算法
- Intra-Connection Strategy
- Inter-Connection Strategy
- 评分标准

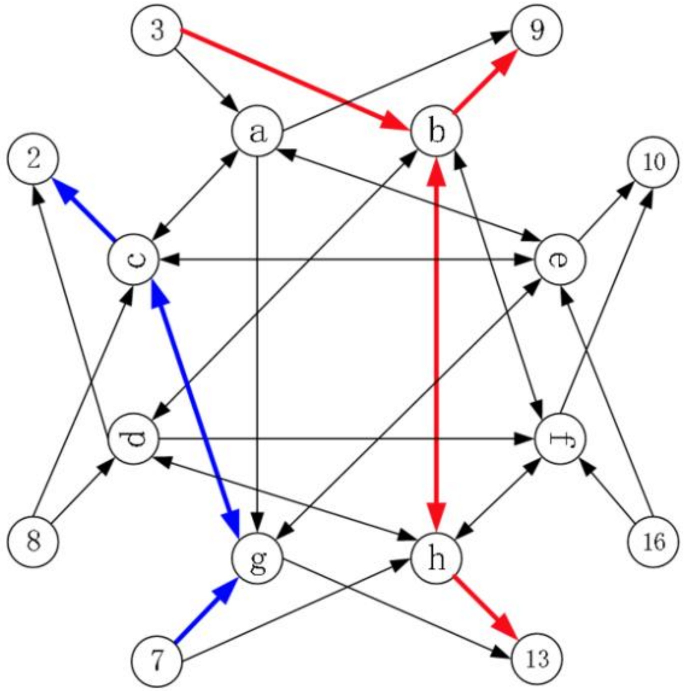
FPGA Routing

(Optimization version) Given a directed graph $G = \langle V, E \rangle$, edge weights $c: E \to R^+$, and a collection $M \subseteq V \times 2^V$ of source–destination pairs. Find the node-disjoint subgraphs $G_m = \langle V_m, E_m \rangle \subseteq G$ for all $m \in M$ and minimize $\sum_{m \in M} \sum_{e \in E_m} c(e)$.



FPGA Architecture and

Signals to be Routed



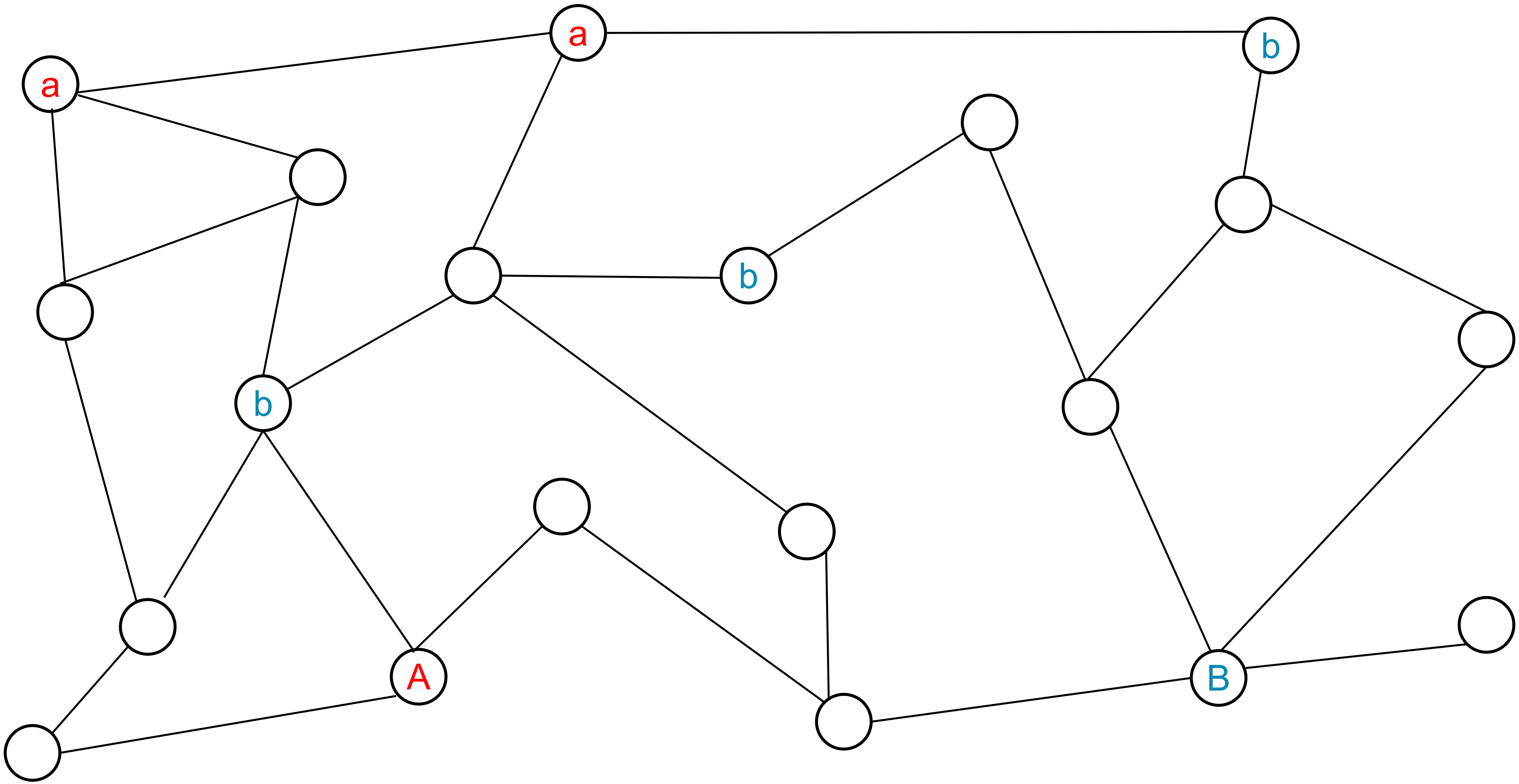
Routing Graph G and

$M = \{ \ (7, \{2\}) \ , \ (3, \{9, 13\}) \ \}$

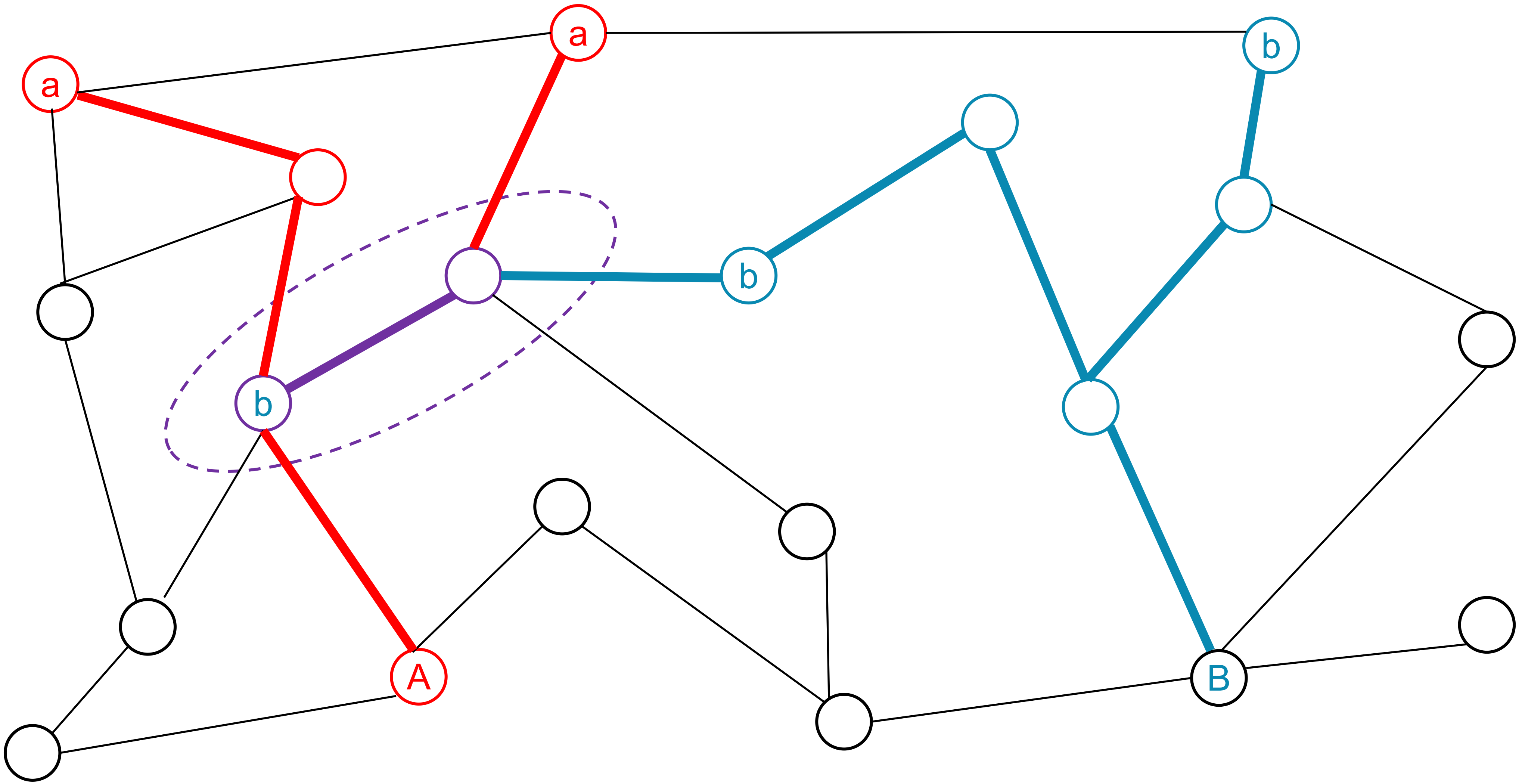
基本概念

- Connection
  – 单个 source 到 单个 sink 的连接路径
- Net
  – 由多条 connection组成
  – 每条 connection 的 source 相同，sink 不同
  – 同一 Net 中的不同 connection 可以共享节点
  – 不同 Net 中的 connection 不可以共享节点
- Node
  – 一条物理的线，可以抽象成图中的一个节点
- Routing 目标
  – 寻找到一组 connection 方案，使所有 connection cost 的最大值尽可能小

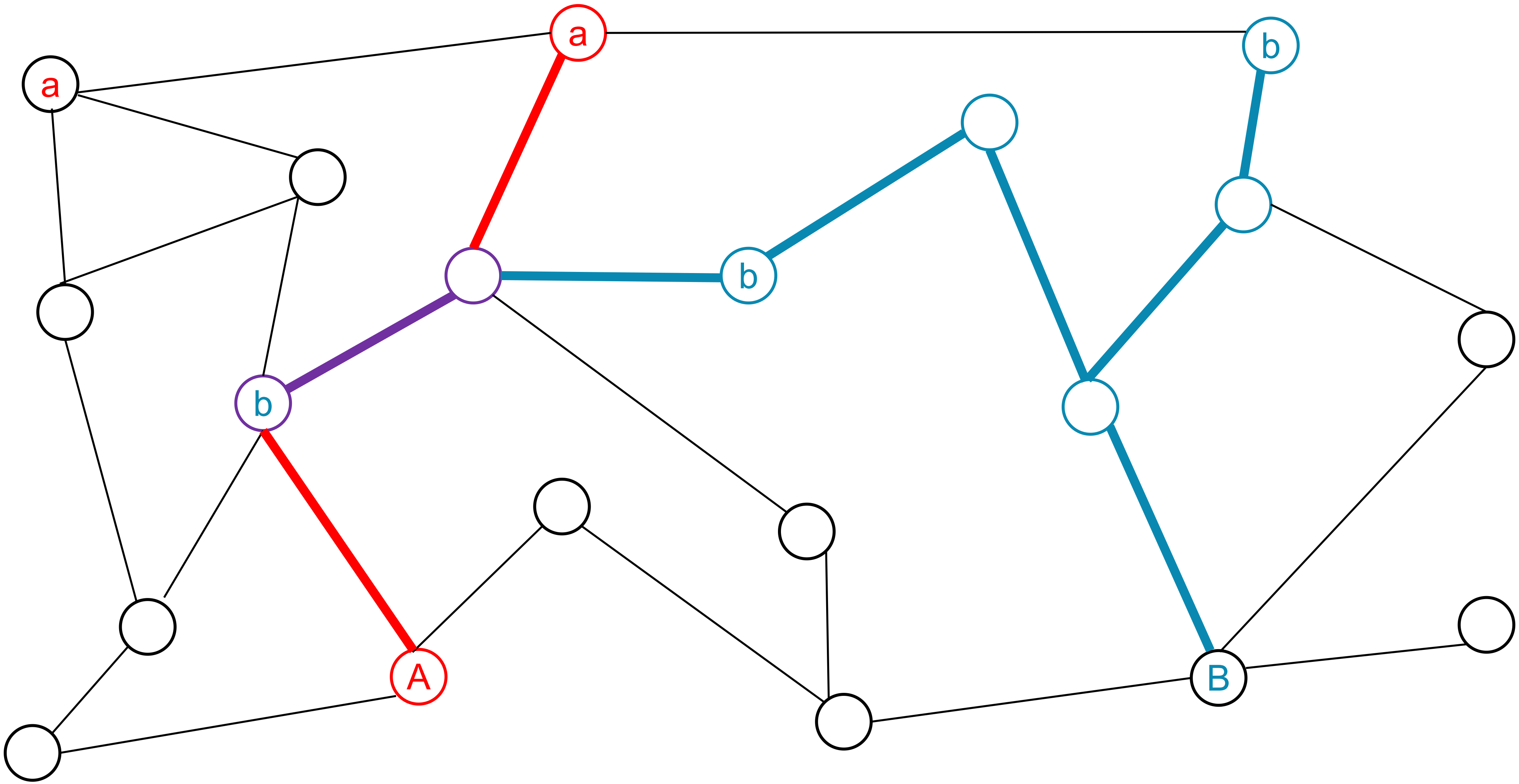
Problem Statement



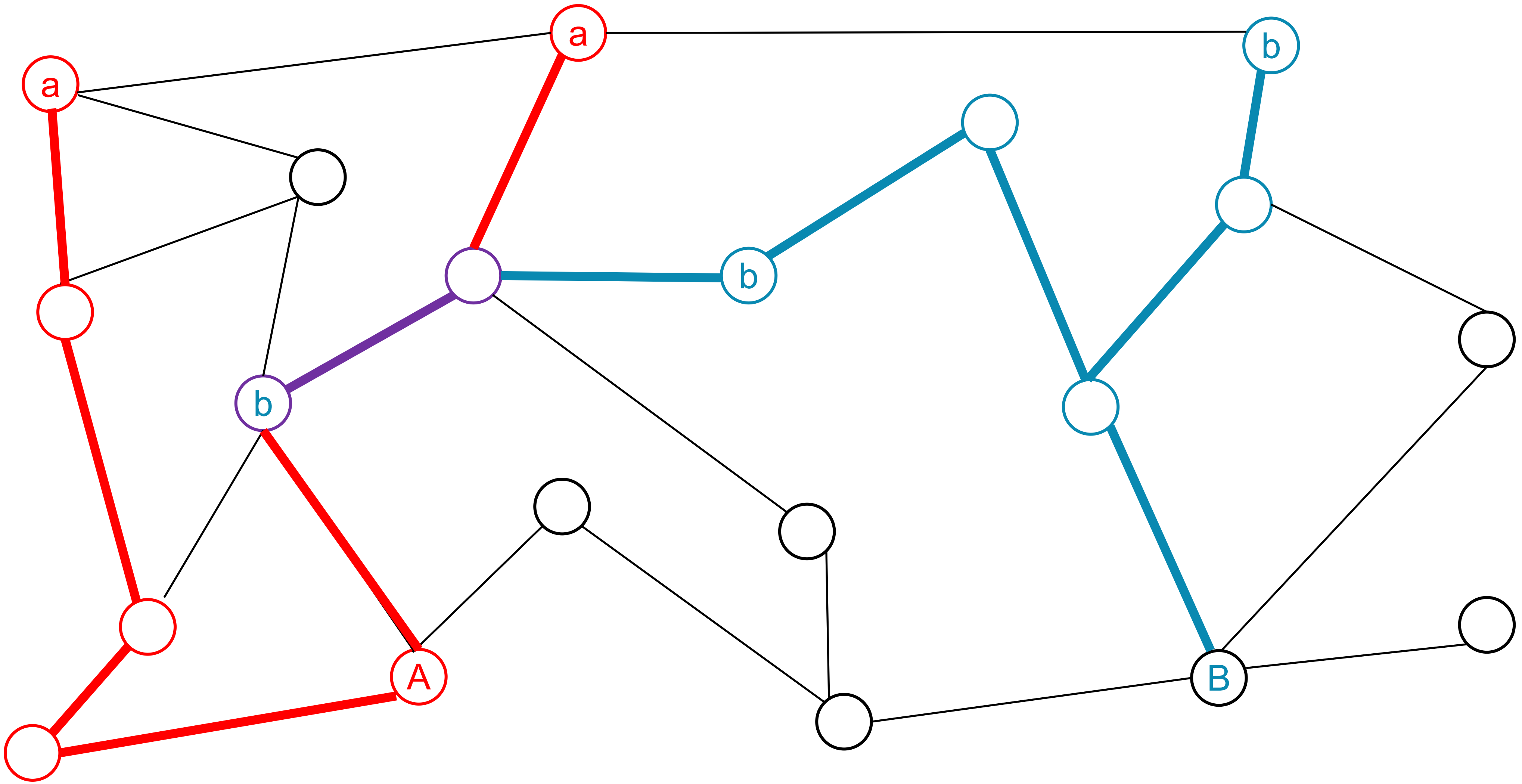
Negotiation Algorithm



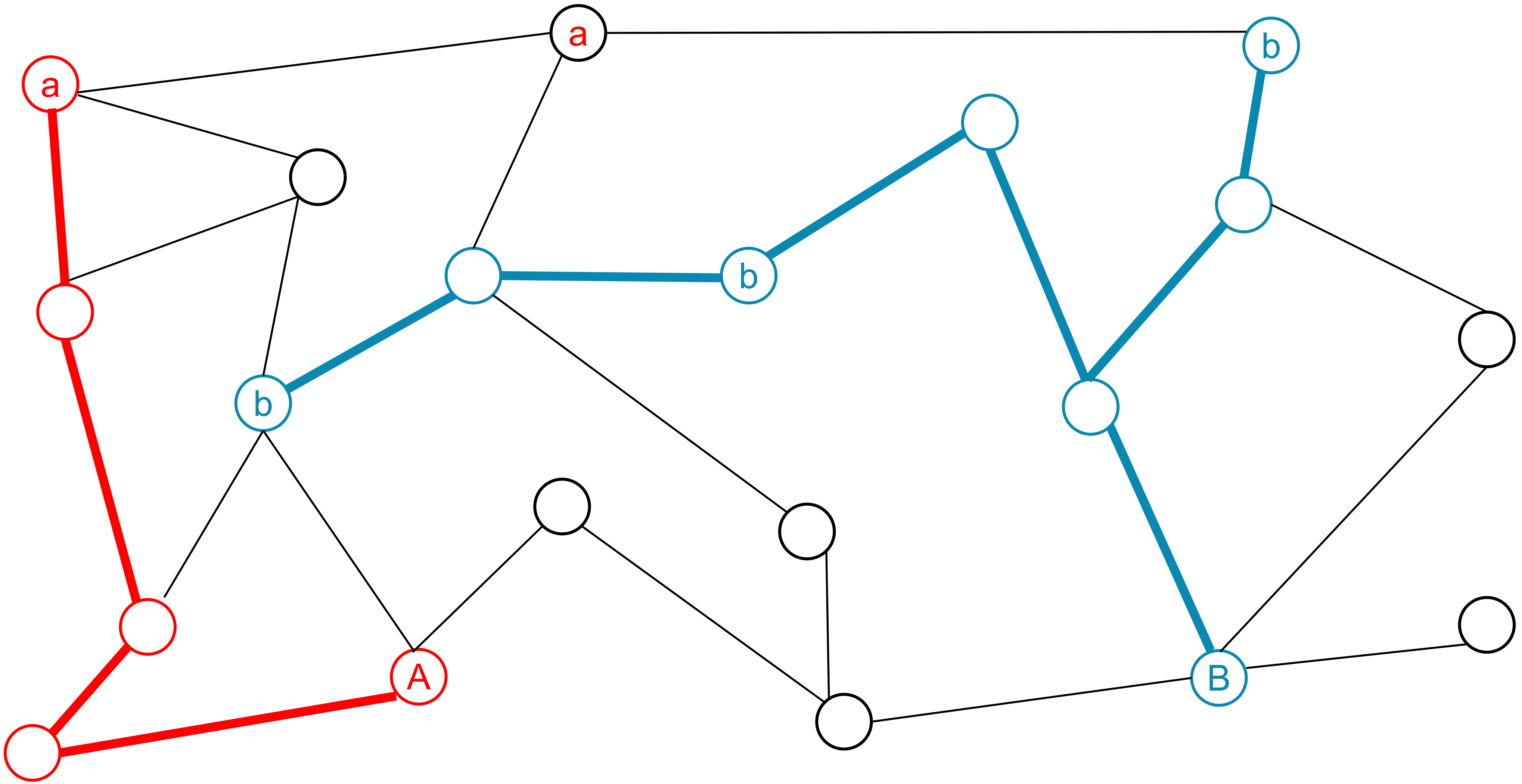
Negotiation Algorithm



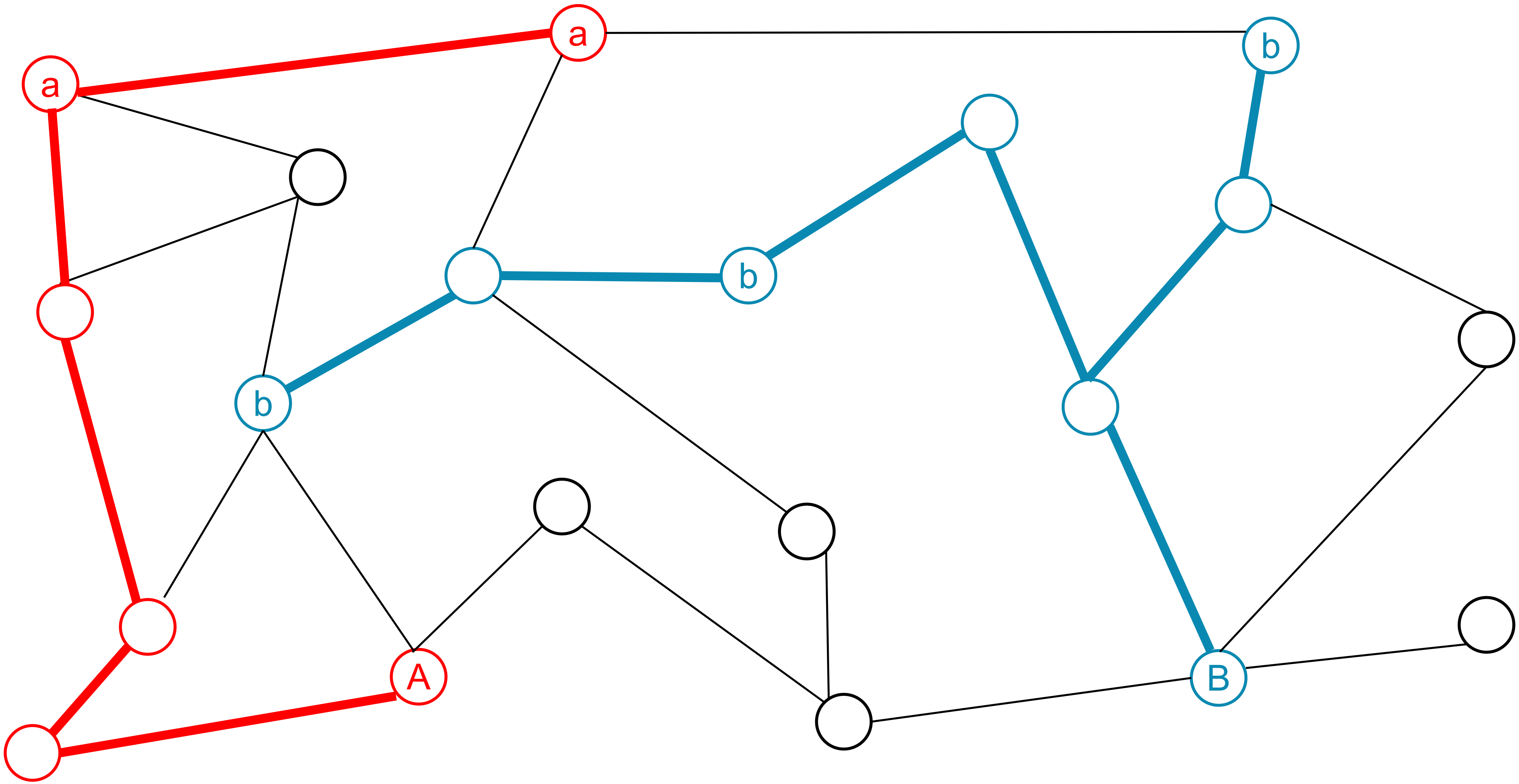
Negotiation Algorithm



Negotiation Algorithm



Negotiation Algorithm



A* Basics

- Best-first search
  - Closed/Open set: nodes that has/has not been expanded.
- Cost

$$f(n) = g(n) + \mathrm{h}(n) \text{ for node } n$$

Cost of path from source to n

Heuristic function of estimated cheapest-cost path from n to the goal

- **Admissible**: h(n) never overestimates the actual cost
- **Monotonic (Consistent):** h(n) ≤ c(n, n0) + h(n0) for each n0 being a successor of n

A* Basics

Algorithm 1: A*

Initialize OPEN to $\{s_0\}$;
while $OPEN \neq \emptyset$ **do**
    Get and remove from OPEN a node n with a smallest $f(n)$;
    Add n to CLOSED;
    if *n is a goal node* **then**
        Return solution path from s_0 to n;
    for *every successor n' of n* **do**
        $g_1 = g(n) + c(n, n')$;
        if $n' \in CLOSED$ **then**
            if $g_1 < g(n')$ **then**
                Remove n' from CLOSED and add it to OPEN;
            else
                Continue;
        else
            if $n' \notin OPEN$ **then**
                Add n' to OPEN;
            else if $g_1 \geq g(n')$ **then**
                Continue;
        Set $g(n') = g_1$;
        Set $f(n') = g(n') + h(n')$;
        Set parent$(n') = n$;
Return failure (no path exists);

PathFinder

- 我们已实现基于 A* 搜索的串行 PathFinder
  - *fpga-route/src/ace-route.cc/path_finder()*

Algorithm 5: Pseudo-code of the Pathfinder routing algorithm [31]

```
Let: RT_i be the set of nodes in the current routing of net i
while shared resources exist do
    /*Illegal routing*/
    foreach net, i do
        rip-up routing tree RT_i;
        RT(i) = s_i foreach sink t_ij do
            Initialize priority queue PQ to RT_i at cost 0;
            while sink t_ij not found do
                Remove lowest cost node m from PQ;
                foreach fanout node n of node m do
                    Add n to PQ at PathCost(n) = c_n + PathCost(m);
                end
            end
            foreach node n in path t_ij to s_i do
                /*backtrace*/
                Update c_n;
                Add n to RT_i;
            end
        end
    end
    update h_n for all n;
end
```

Strategy 1 : Intra-Connection Parallelization

- 在单个 connection 寻路过程中并行搜索
- *Parallel A**
- *Hash Distributed A**
- *Parallel Bidirectional A**

Parallel A* Paradigms

- **Centralized**: Simple Parallel A* (SI
  - Share an open list among threads
  - Straightforward, Easy to implement.

What’s the problem?

Algorithm 2: Simple Parallel A* (SPA*)

```
1  Initialize OPEN_shared to {s_0};
2  Initialize Lock l_o, l_i;
3  Initialize incumbent.cost = ∞;
4  In parallel, on each thread, execute 5-32;
5  while TerminateDetection() do
6      if OPEN_shared = ∅ or Smallest f(n) value of n ∈ OPEN_shared ≥ incumbent.cost then
7          Continue;
8      AcquireLock(l_o);
9      Get and remove from OPEN_shared a node n with a smallest f(n);
10     ReleaseLock(l_o);
11     Add n to CLOSED_shared ;
12     if n is a goal node then
13         AcquireLock(l_i);
14         if path cost from s_0 to n < incumbent.cost then
15             incumbent = path from s_0 to n;
16             incumbent.cost = path cost from s_0 to n;
17         ReleaseLock(l_i);
18     for every successor n' of n do
19         g_1 = g(n) + c(n, n');
20         if n' ∈ CLOSED_shared then
21             if g_1 < g(n') then
22                 Remove n' from CLOSED_shared and add it to OPEN_shared ;
23             else
24                 Continue;
25         else
26             if n' ∉ OPEN_shared then
27                 Add n' to OPEN_shared ;
28             else if g_1 ≥ g(n') then
29                 Continue;
30         Set g(n') = g_1;
31         Set f(n') = g(n') + h(n');
32         Set parent(n') = n;
33 if incumbent.cost = ∞ then
34     Return failure (no path exists);
35 else
36     Return solution path from s_0 to n;
```

Parallel A* Paradigms

- **Decentralized:**
  - Each thread has its own open list

    What's the problem?

  - Load balancing
  - Termination detection
  - Duplication elimination

Algorithm 3: Decentralized A* with Local OPEN/CLOSED lists

1 Initialize $OPEN_p$ for each thread p;
2 Initialize *incumbent.cost* $= \infty$;
3 Add s_0 to $OPEN_{ComputeRecipient(s_0)}$;
4 In parallel, on each thread p, execute 5-31;
5 **while** *TerminateDetection()* **do**
6   **while** $BUFFER_p \neq \emptyset$ **do**
7   Get and remove from $BUFFER_p$ a triplet (n', g_1, n);
8   **if** $n' \in CLOSED_p$ **then**
9   **if** $g_1 < g(n')$ **then**
10   Remove n' from $CLOSED_p$ and add it to $OPEN_p$;
11   **else**
12   Continue;
13   **else**
14   **if** $n' \notin OPEN_p$ **then**
15   Add n' to $OPEN_p$;
16   **else if** $g_1 \geq g(n')$ **then**
17   Continue;
18   Set $g(n') = g_1$;
19   Set $f(n') = g(n') + h(n')$;
20   Set parent$(n') = n$;
21   **if** $OPEN_p = \emptyset$ ***or*** *Smallest* $f(n)$ *value of* $n \in OPEN_p \geq$ *incumbent.cost* **then**
22   Continue;
23   Get and remove from $OPEN_p$ a node n with a smallest $f(n)$;
24   Add n to $CLOSED_p$;
25   **if** *n is a goal node* **then**
26   **if** *path cost from s_0 to n* $<$ *incumbent.cost* **then**
27   *incumbent* $=$ path from s_0 to n;
28   *incumbent.cost* $=$ path cost from s_0 to n;
29   **for** *every successor n' of n* **do**
30   Set $g_1 = g(n) + c(n, n')$;
31   Add (n', g_1, n) to $BUFFER_{ComputeRecipient(n)}$;
32 **if** *incumbent.cost* $= \infty$ **then**
33   Return failure (no path exists);
34 **else**
35   Return solution path from s_0 to n;

Hash Distributed A*

HDA* starts by expanding the initial state at the root processor. Then, each processor P executes the following loop until an optimal solution is found:

1. First, P checks whether one or more new states have been received in its message queue. If so, P checks for each new state s in $Closed_P$, in order to determine whether s is a duplicate, or whether it should be inserted in $Open_P$.[2]

2. If the message queue is empty, then P selects a highest priority state from $Open_P$ and expands it, resulting in newly generated states. For each newly generated state s, a hash key $K(s)$ is computed based on the state representation, and the re$K(s)$ and s is sent to the processor that owns $K(s)$. This send is asynchronous and non-blocking. P continues its computation without waiting for a reply from the destination.

Parallel Bidirectional A*

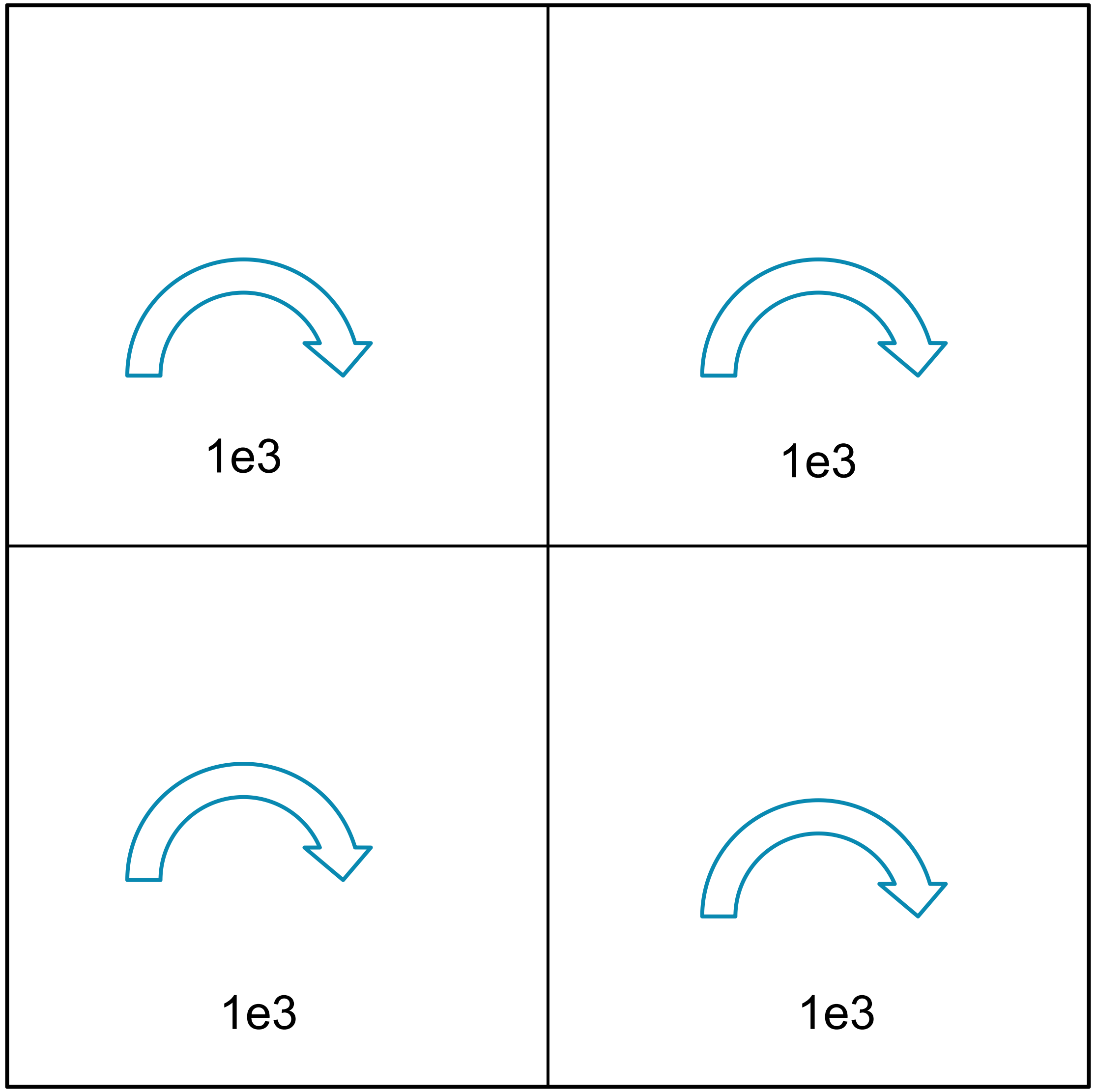
Algorithm 1 PNBA*, Parallel New Bidirectional A*

while ¬finished **do**
  $x \leftarrow \text{open}_1.\text{pop}()$
  if $x \in \mathcal{M}$ **then**
    if $\left(f_1(x) < \mathcal{L}\right) \wedge \left(g_1(x) + F_2 - h_2(x) < \mathcal{L}\right)$ **then**
      for all edges (x, y) of the graph being explored **do**
        if $\left(y \in \mathcal{M}\right) \wedge \left(g_1(y) > g_1(x) + d_1(x, y)\right)$ **then**
          $g_1(y) \leftarrow g_1(x) + d_1(x, y)$
          $f_1(y) \leftarrow g_1(y) + h_1(y)$
          if $y \in \text{open}_1$ **then**
            $\text{open}_1.\text{remove}(y)$
          $\text{open}_1.\text{insert}(y)$
          if $g_1(y) + g_2(y) < \mathcal{L}$ **then**
            lock
            if $g_1(y) + g_2(y) < \mathcal{L}$ **then**
              $\mathcal{L} \leftarrow g_1(y) + g_2(y)$
            unlock
    $\mathcal{M} \leftarrow \mathcal{M} - \{x\}$
  if $\text{open}_1.\text{size}() > 0$ **then**
    $F_1 \leftarrow f_1(\text{open}_1.\text{peek}())$
  else
    finished $\leftarrow$ **true**

Strategy 2 : Inter-Connection Parallelization

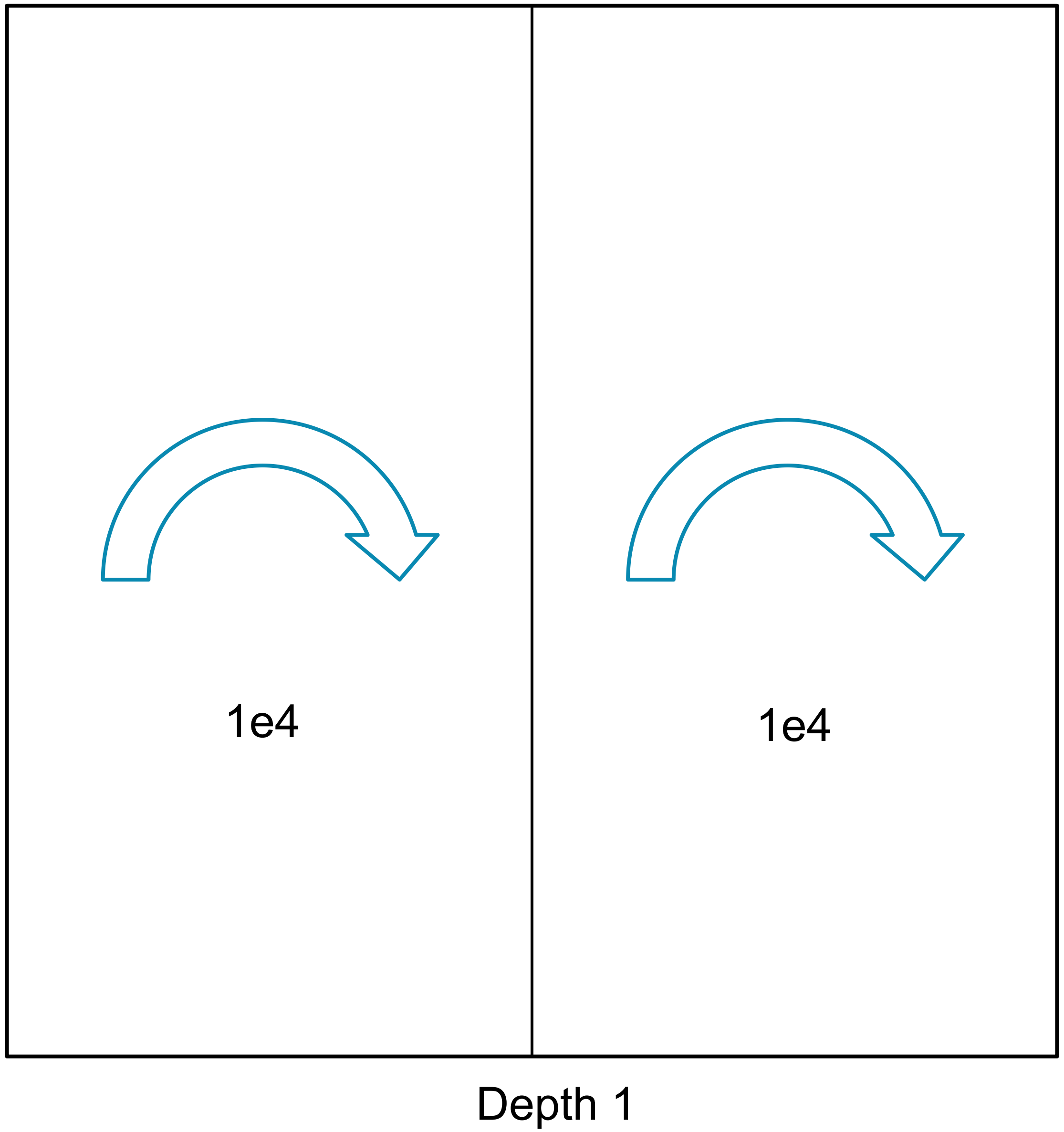
- 不同 connection 并行进行寻路过程
- 按区域划分 connection
- *Intuition Parallel*
- *Dense Parallel*

Intuition Parallel Method

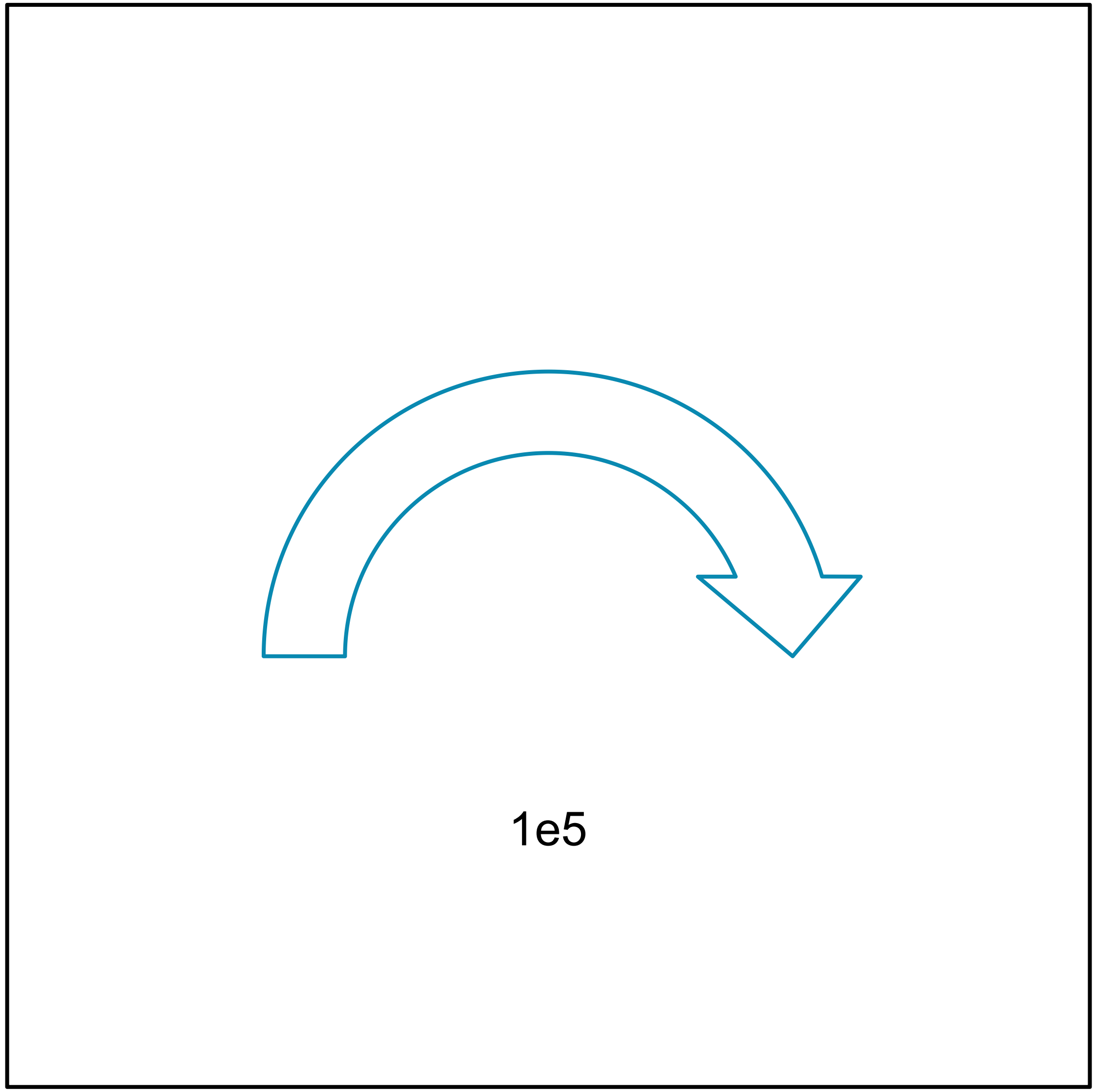


Parallelizable division and conquer based on spatial partitioning
Depth 2

Intuition Parallel Method



Intuition Parallel Method



Depth 0

Intuition Parallel Method

- Parallelism is restricted by overlapped nets.

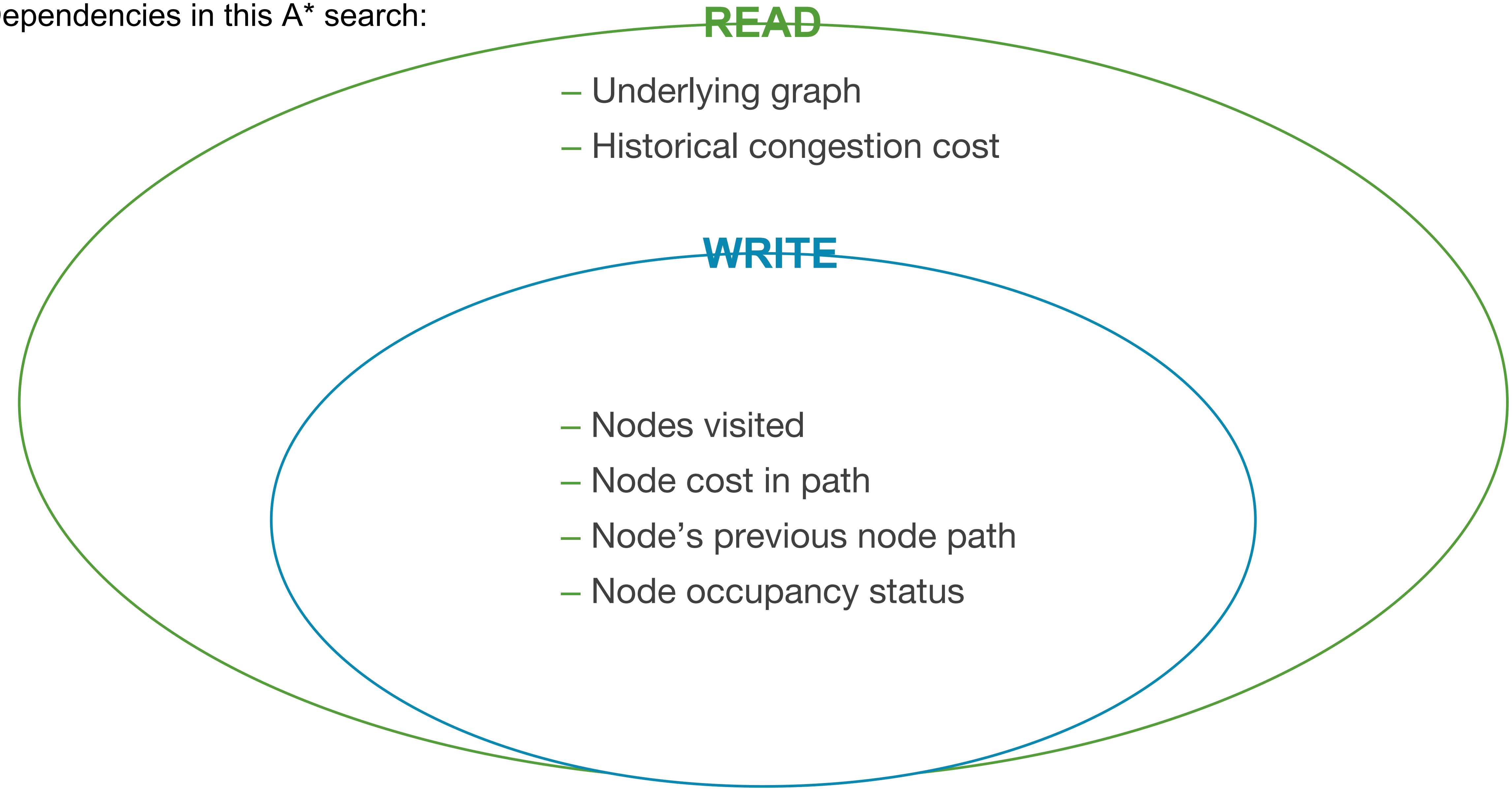
- E.g. $$\frac{1000+10000+100000}{1000*4+10000*2+100000} \approx 89.5\%$$

Dense Parallel

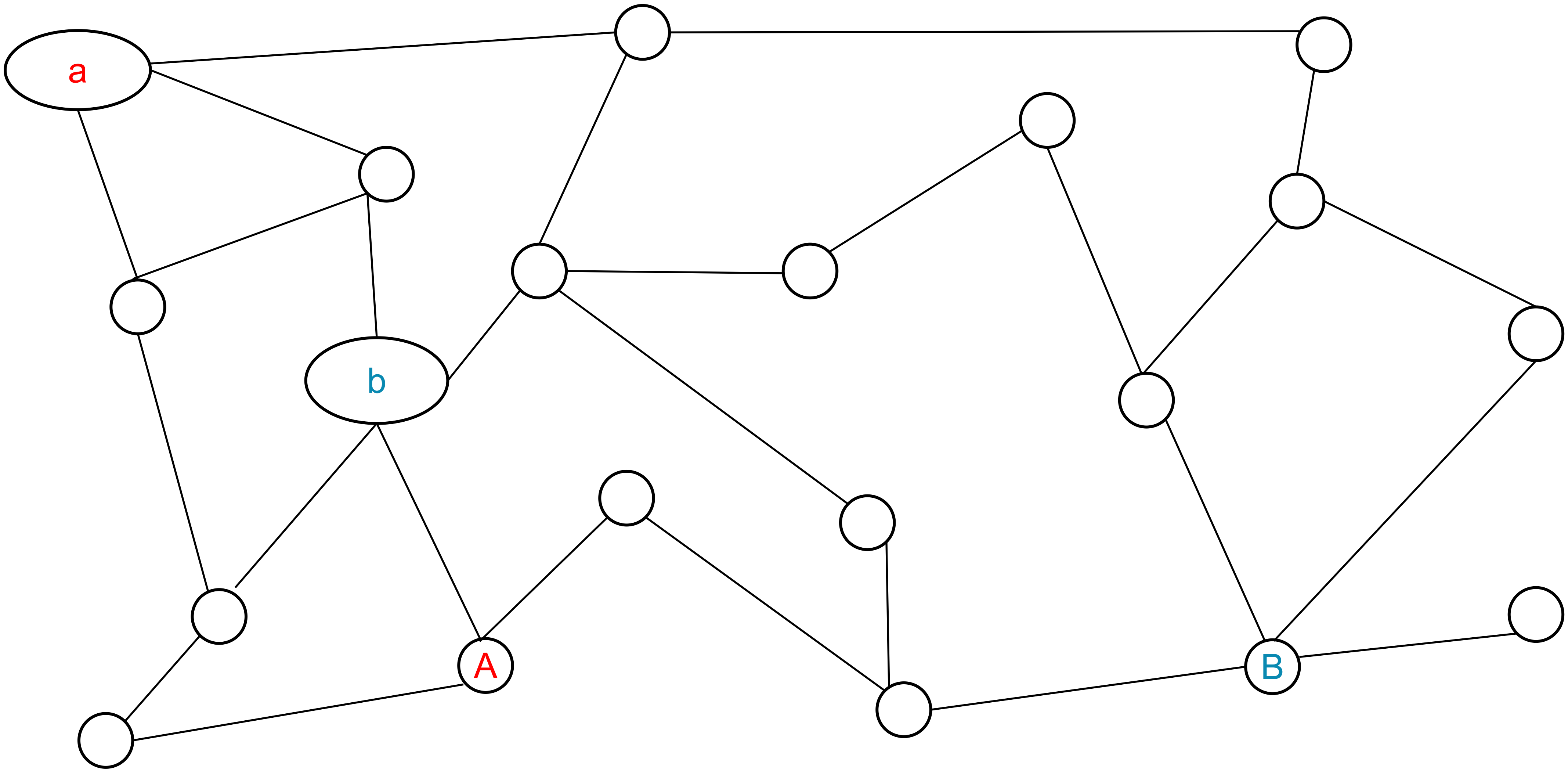
- Idea: Allow Overlapped Nets to be paralleled
- I. Modify underlying data structure and design synchronization mechanism to allow one node to be explored in multiple threads simultaneously
- II. Optimize synchronization mechanism to limit overhead

Part I Parallelism Construction

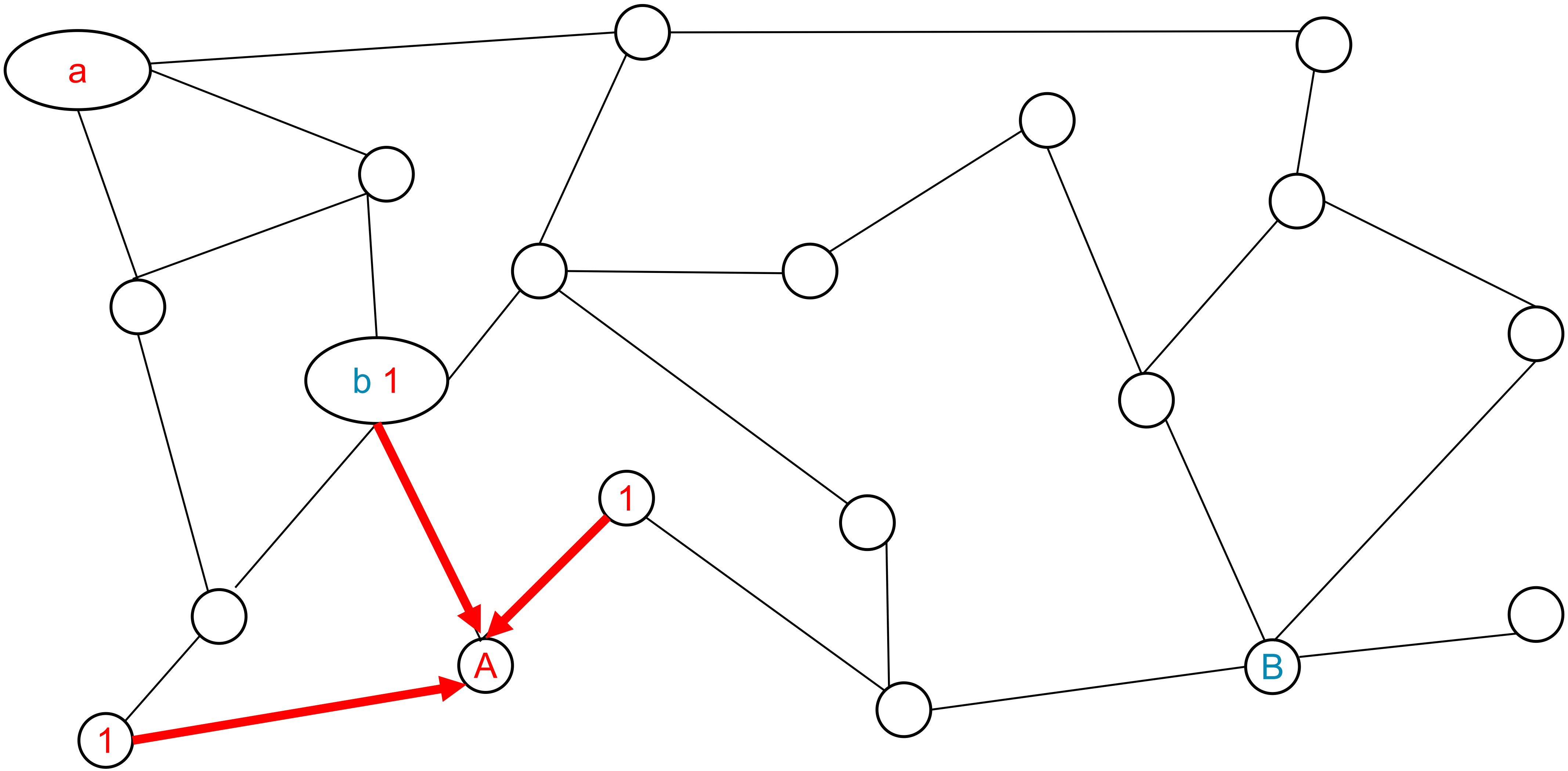
Dependencies in this A* search:



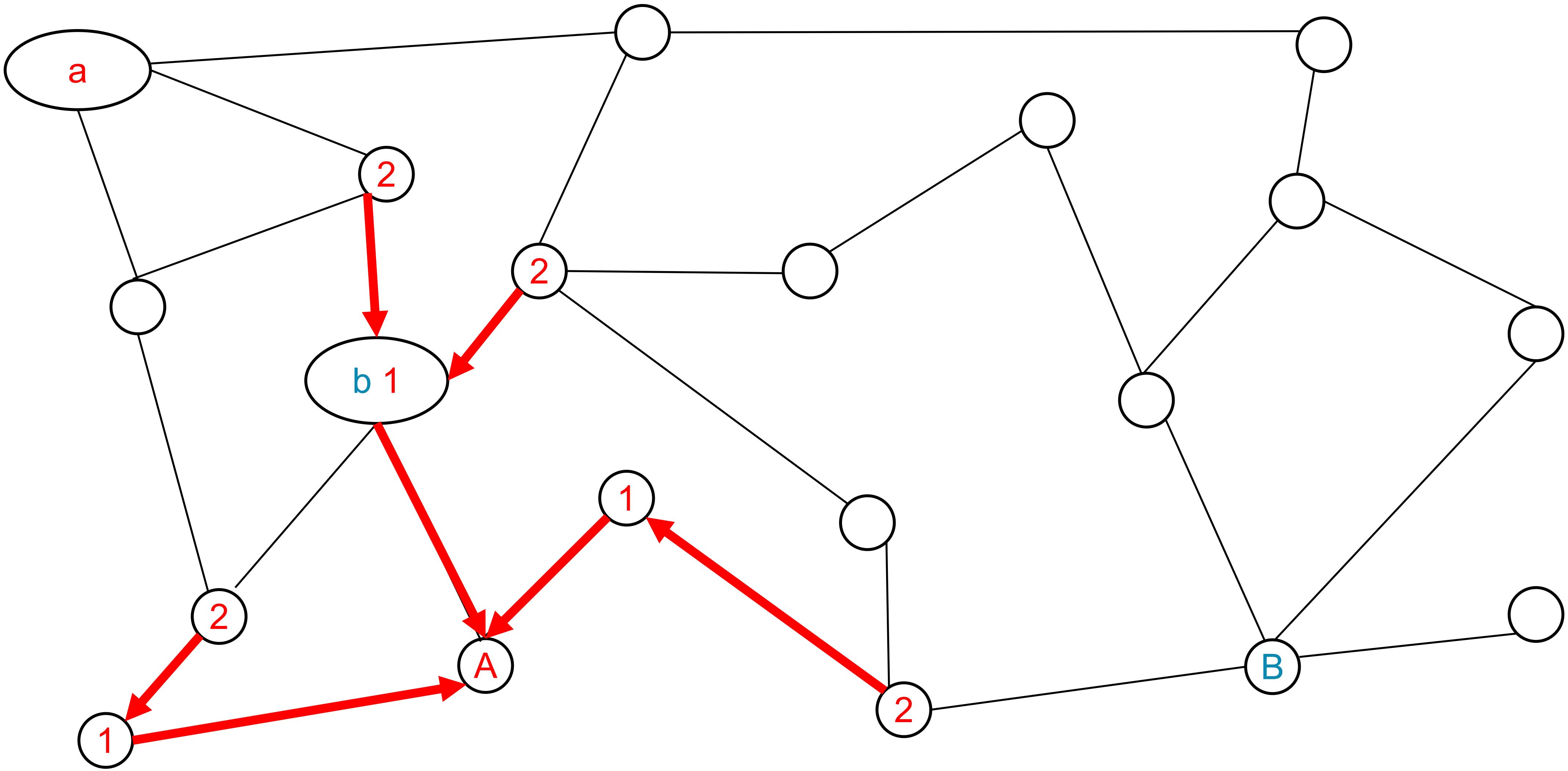
Part I Parallelism Construction



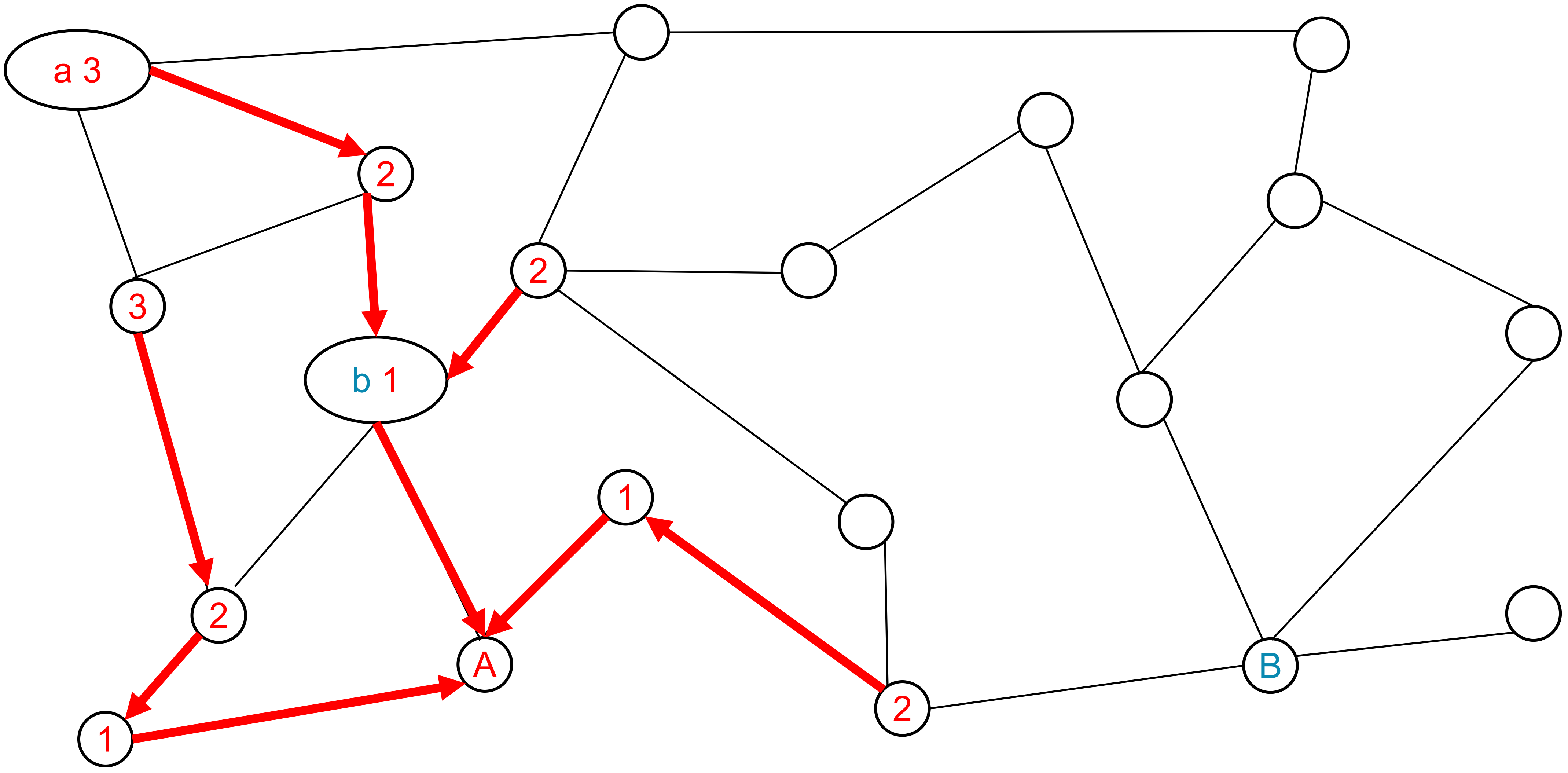
Part I Parallelism Construction



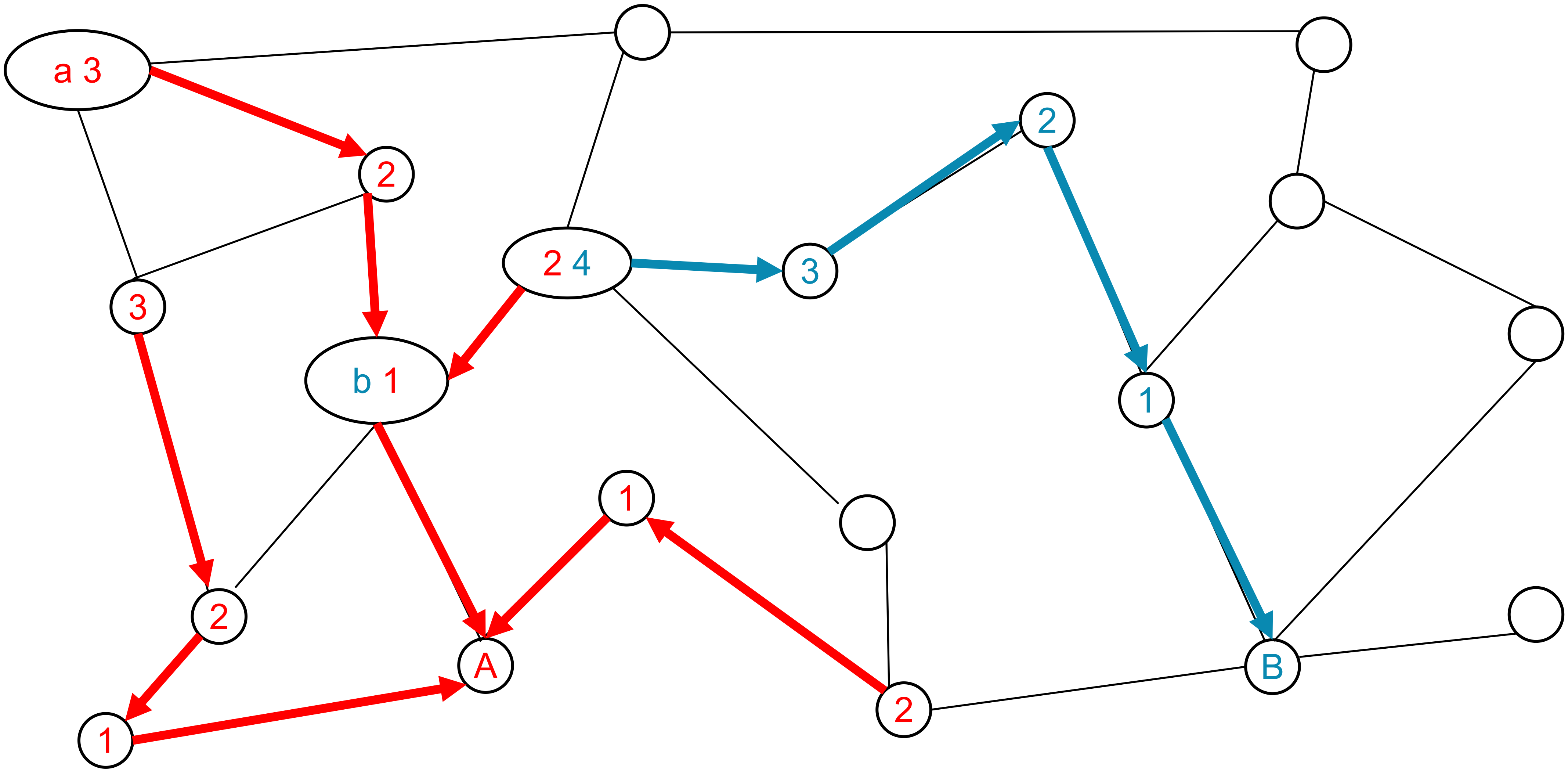
Part I Parallelism Construction



Part I Parallelism Construction



Part I Parallelism Construction



Thread Private: Visited; Cost in Path; Path(Previous Node)

Part I Parallelism Construction

Dependencies in this A* search:

READ

Shared
- Underlying graph
- Historical congestion cost

WRITE

Thread Private
- Nodes visited
- Node cost in path
- Node’s previous node path

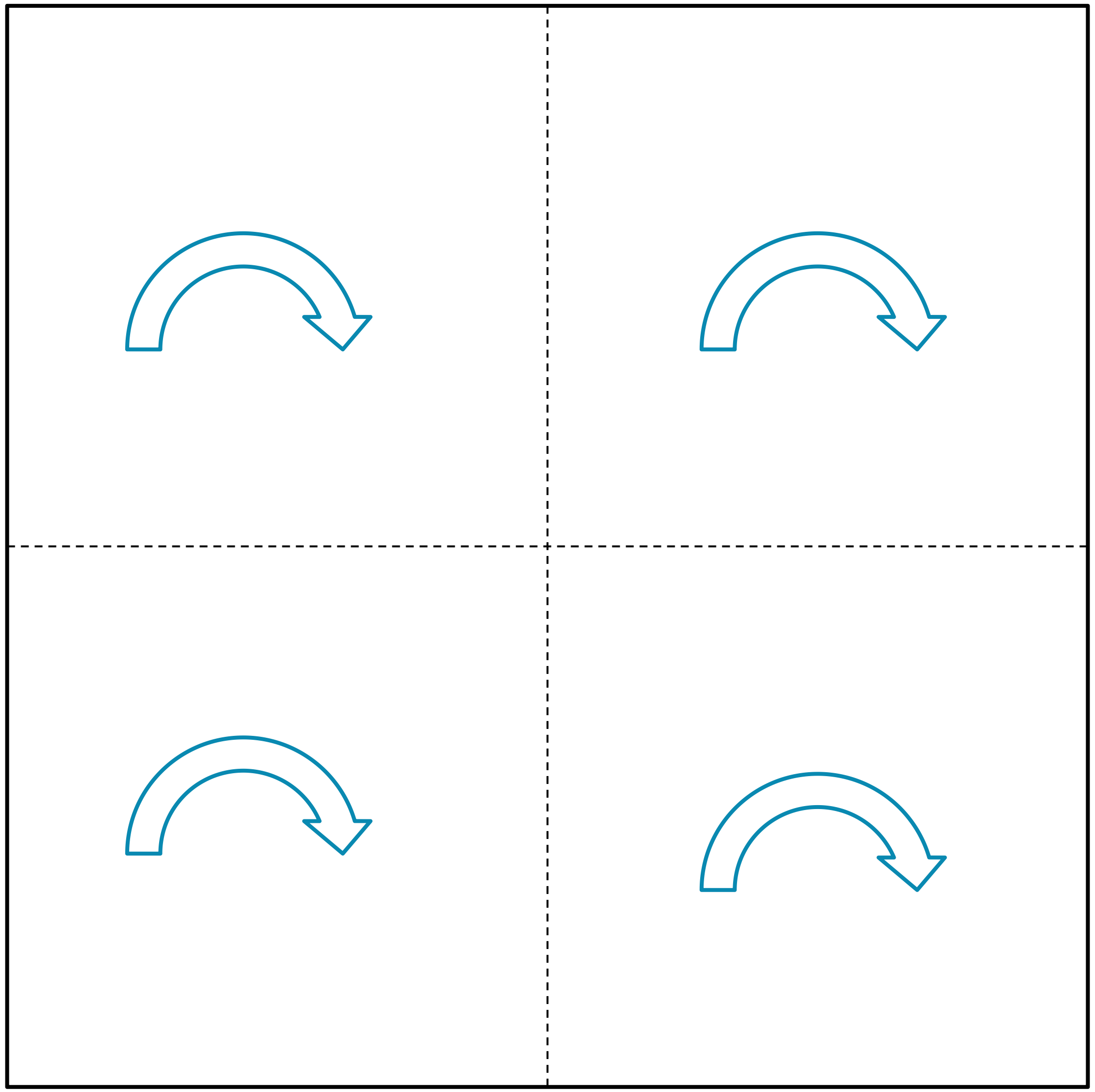
Shared With Lock
- Node occupancy status

Part II Synchronization Optimization

- 1. Sort Strategy
- 2. Atomic Built flag
- 3. Dynamic bunch node allocator
- 4. Lock-Free No-Delete hash map

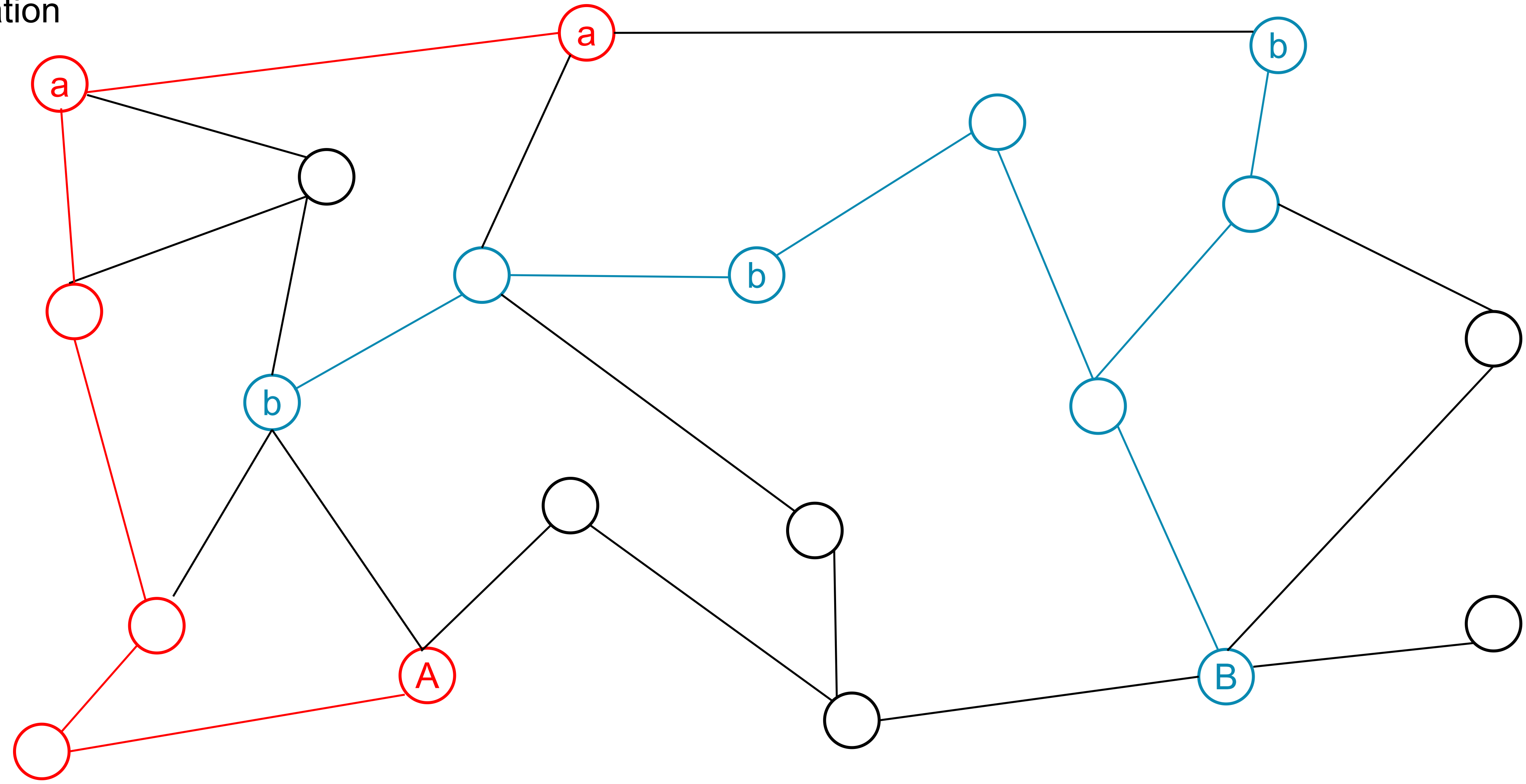
Part II

1. Sort Strategy



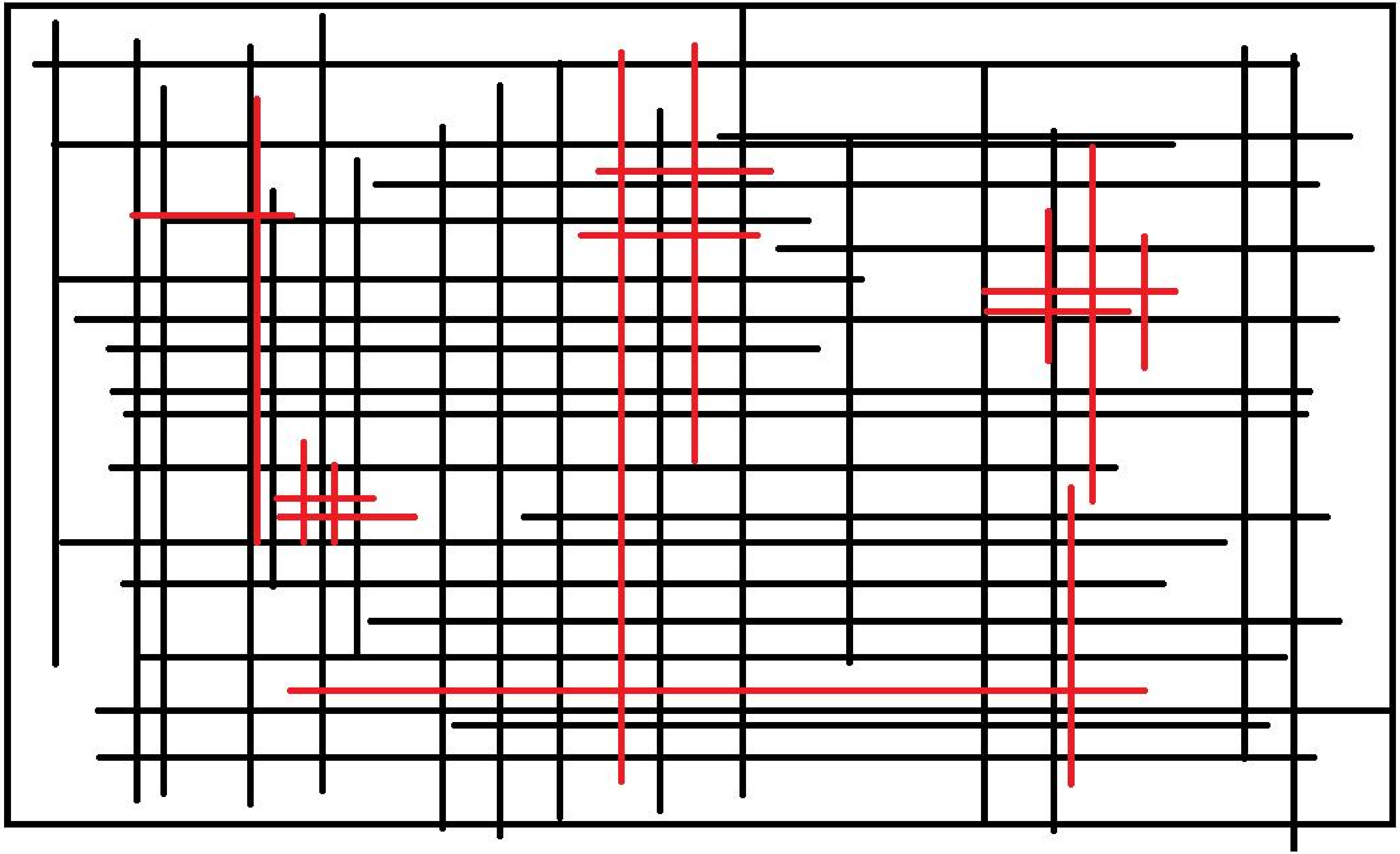
Part II 2. Atomic Built flag

In Imagination



Part II 2. Atomic Built flag

In Fact



Build real nodes and edges dynamically to cut branches and save construction time

Part II 2. Atomic Built flag

Dependencies in this A* search:

READ

Shared
- Historical congestion cost
- Underlying graph

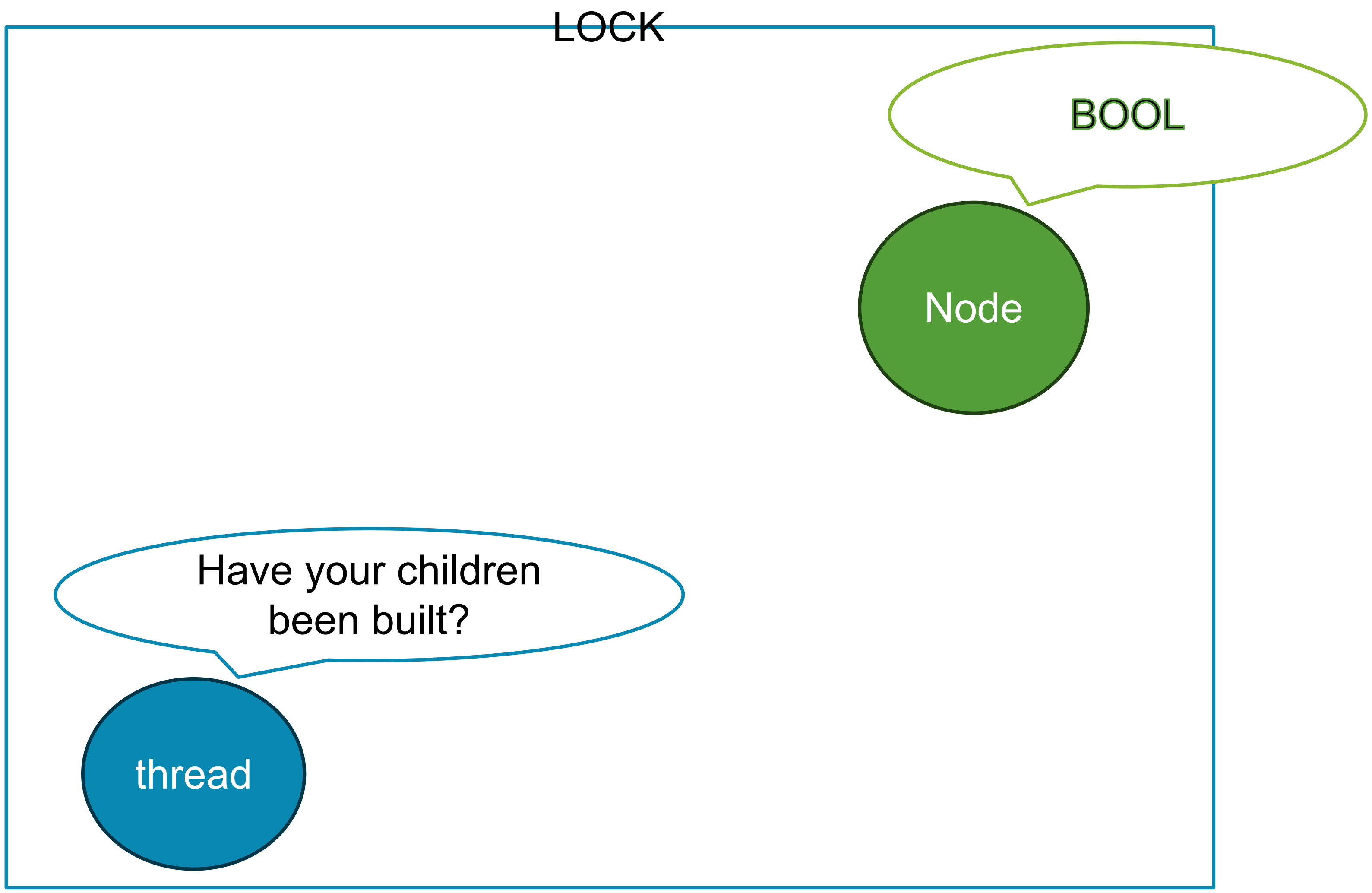
WRITE

Thread Private
- Nodes visited
- Node cost in path
- Node's previous node path

Shared With Lock
- Node occupancy status

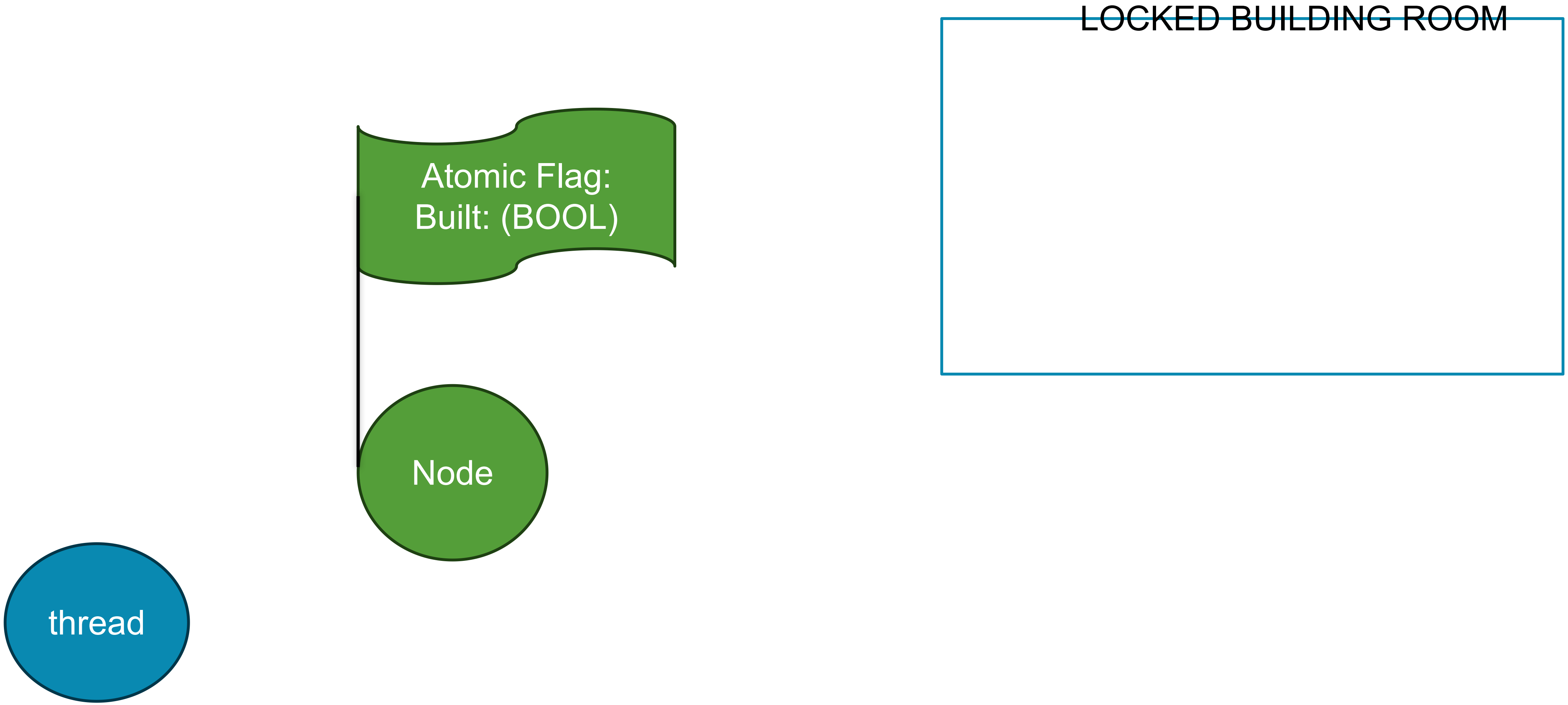
Part II

2. Atomic Built flag

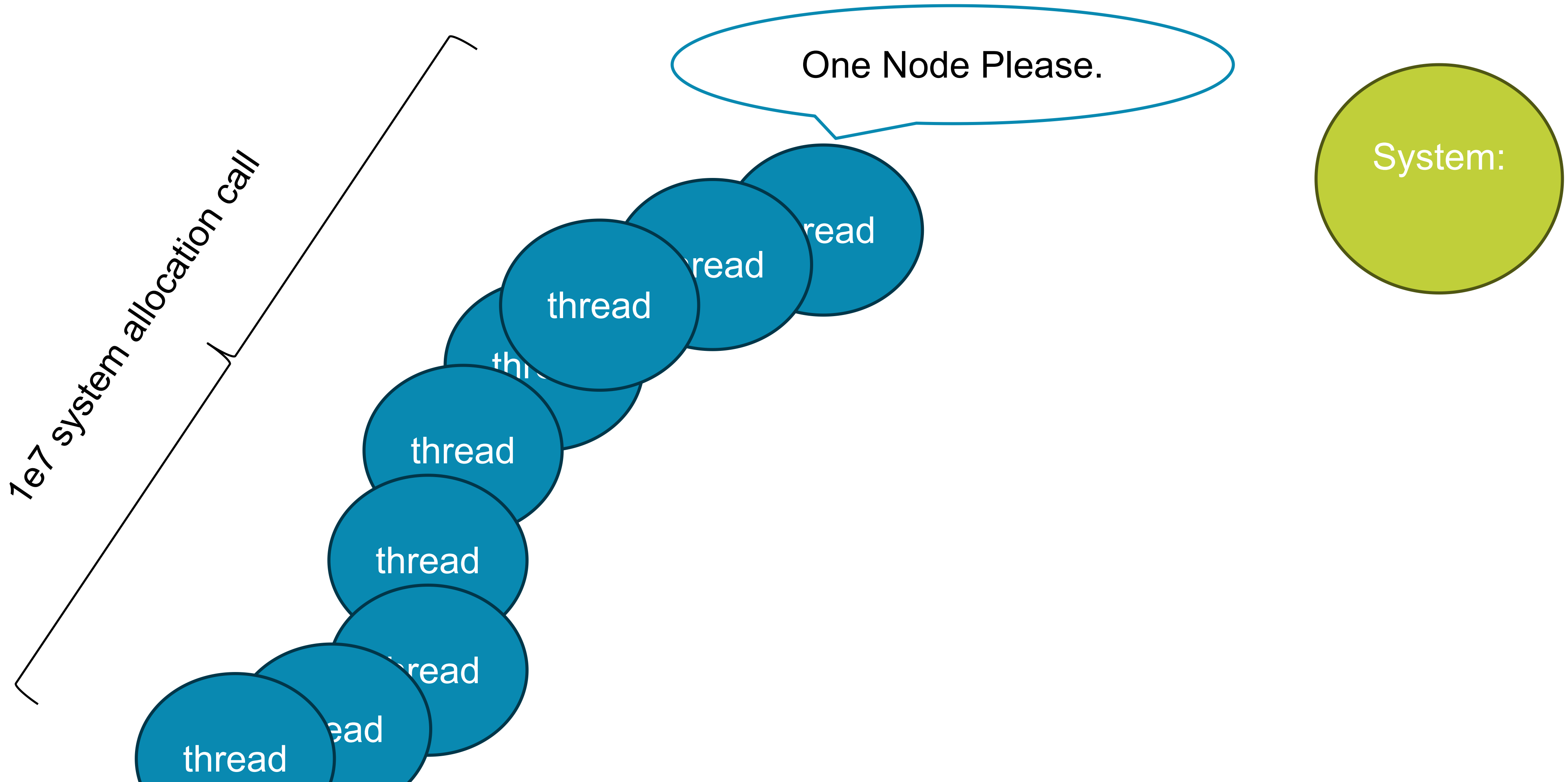


Part II

2. Atomic Built flag

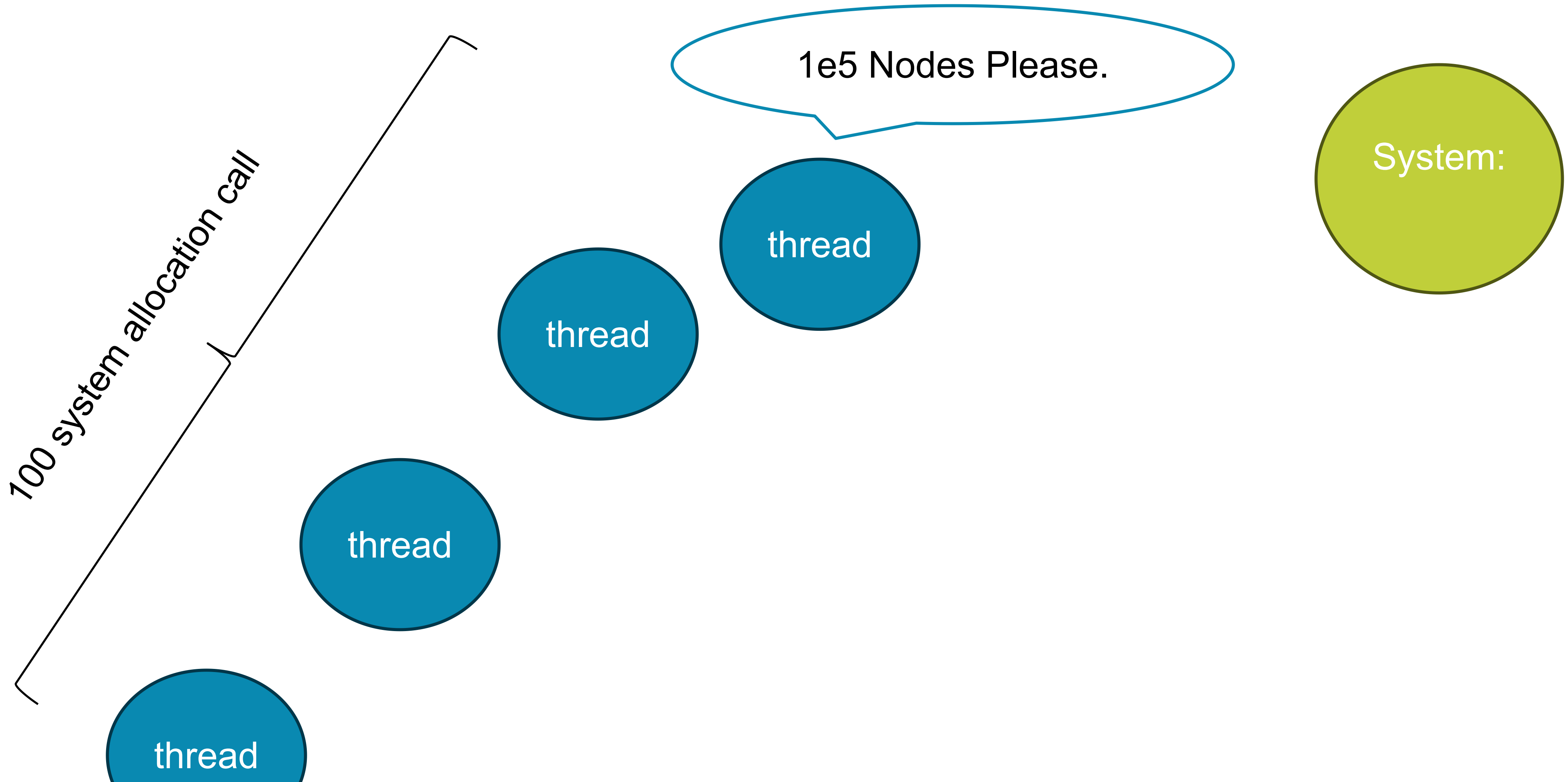


Part II 3. Dynamic bunch node allocator

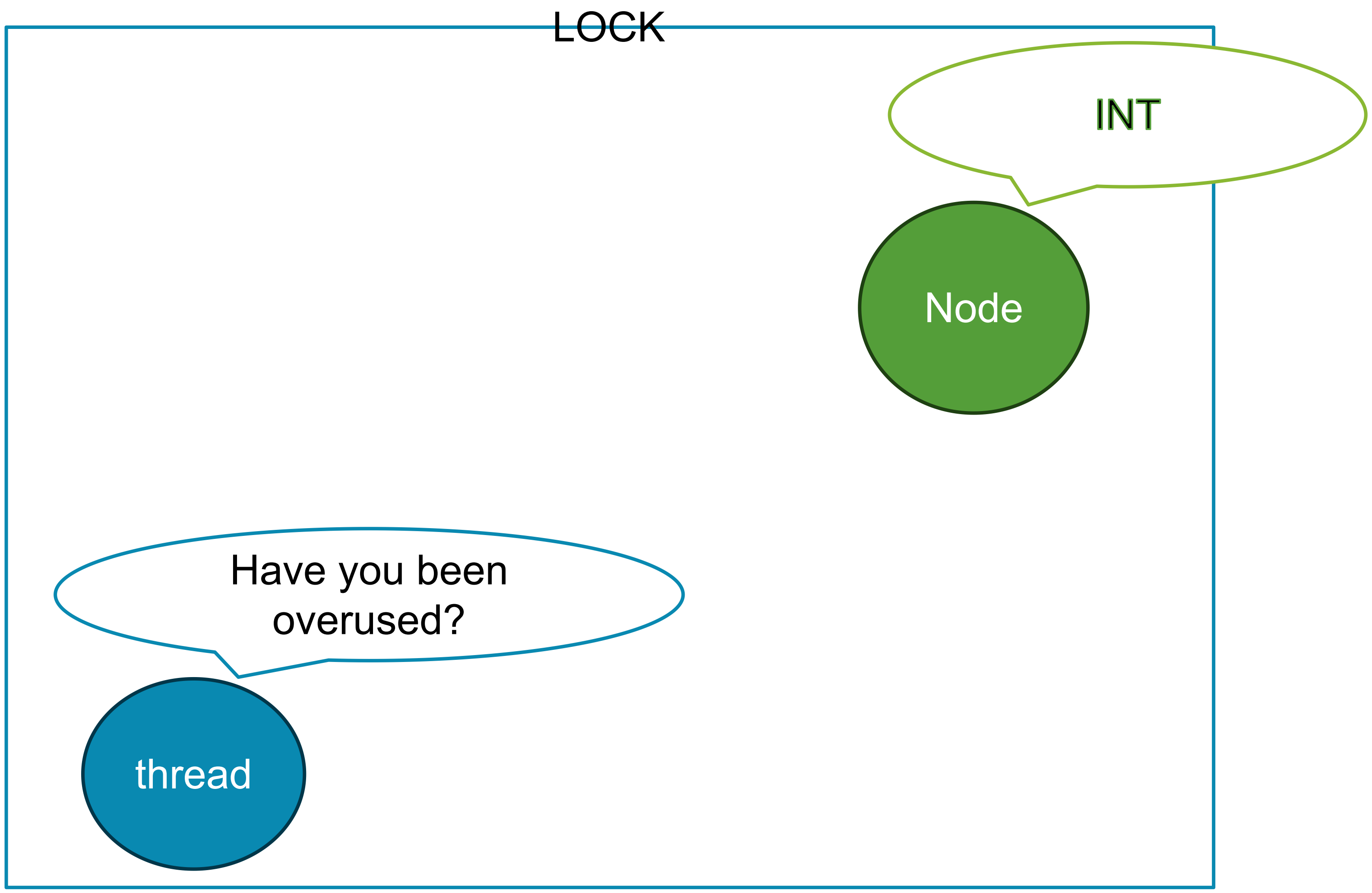


Part II 3. Dynamic bunch node allocator

After Implementing a thread private node allocator

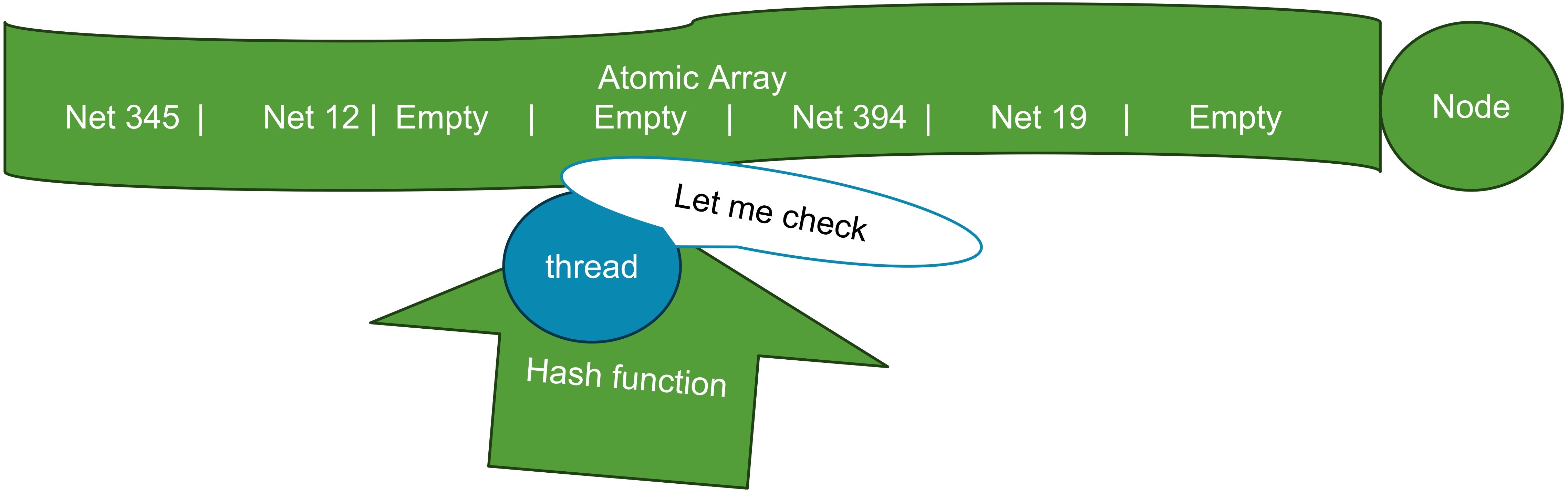


Part II 4.Lock-Free No-Delete hash map



Part II 4.Lock-Free No-Delete hash map

LOCK



Part II Synchronization Optimization

Under 32 Threads:

- 1.Sort Strategy
- 80%->60%

- 2. Atomic Built flag
- 60%->40%

- 3. Dynamic bunch node allocator
- 40%->10%

- 4. Lock-Free No-Delete hash map
- 10%->5%

Result Example (Bidirectional Parallel A*)

Table 2: Critical-path wirelength normalized with RWRoute across all designs (min/max/mean) for different routers.

		Min	Max	Mean
Base Routers	RWRoute	1.00	1.00	1.00
	Vivado	0.83	1.51	1.12
Intra-Connection Opt. Only	Basic	0.91	1.23	1.03
	BE (Serial)	0.82	1.26	1.02
	BE (Parallel)	0.89	1.62	1.08
	Adaptive	0.92	1.25	1.04
Inter-Connection Par. × Intra-Connection Opt.	Basic	0.83	1.64	1.03
	BE (Parallel)	0.90	1.26	1.04
	Adaptive	0.82	1.27	1.01

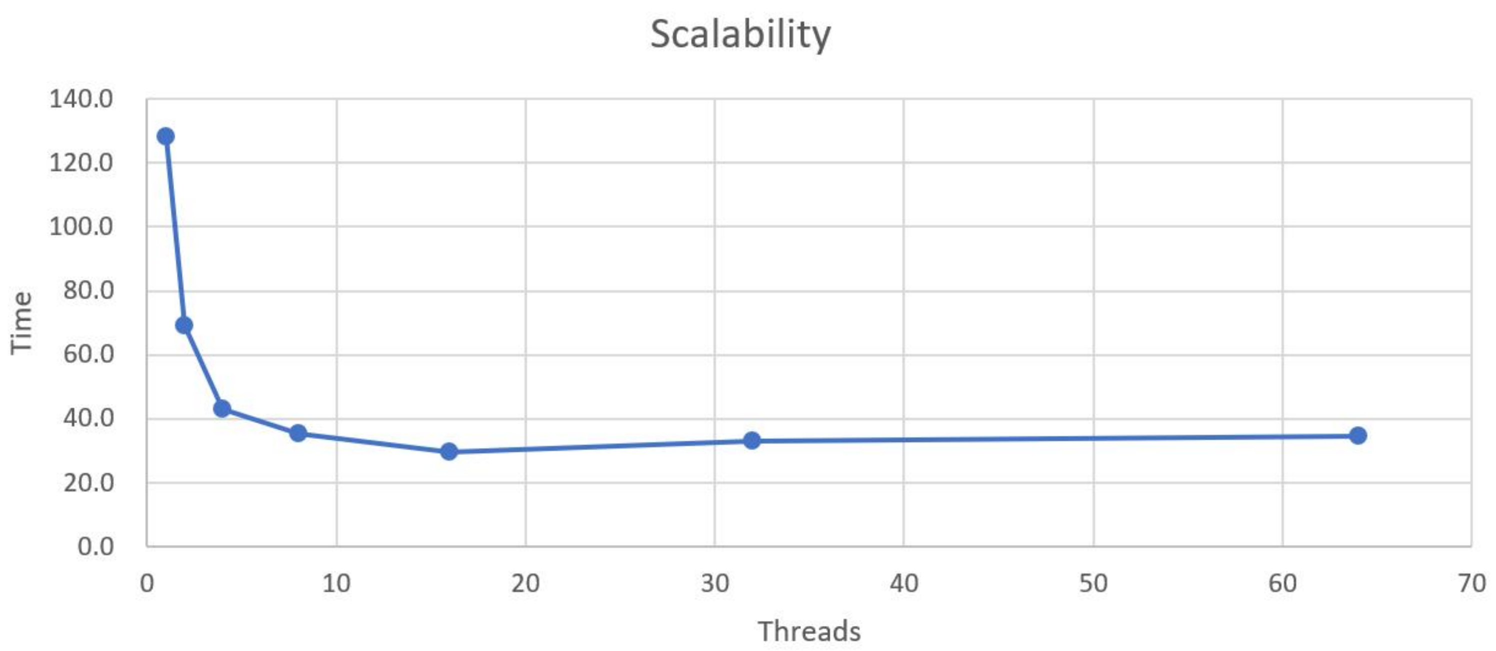
Table 3: Accelerated inter-connection parallelism by intra-connection optimizations, with speedup over RWRoute.

Bi-partition Par. ×	Basic Speedup	BE (2-thread) Runtime Var.	BE (2-thread) Speedup	Adapt Speedup
boom_med_pb	1.41×	(-5.8%, +9.1%)	3.48×	2.87×
boom_soc	1.61×	(-11.7%, +10.8%)	3.93×	3.74×
boom_soc_v2	1.94×	(-5.8%, +4.7%)	6.90×	6.19×
corescore_500	2.86×	(-4.3%, +3.8%)	3.65×	2.93×
corescore_500_pb	2.57×	(-12.0%, +8.3%)	4.11×	3.47×
corescore_1700	3.26×	(-3.5%, +3.9%)	8.02×	6.64×
corundum_25g	1.90×	(-1.3%, +3.8%)	2.75×	2.21×
ispd16_example2	3.01×	(-6.4%, +12.7%)	2.61×	2.63×
koios_dla_large	3.57×	(-1.1%, +1.4%)	4.25×	3.31×
mlcad_d181	1.23×	(-8.0%, +11.8%)	6.13×	6.09×
mlcad_d181_left...	1.59×	(-6.1%, +5.5%)	9.80×	10.23×
rosetta_fd	1.46×	(-3.6%, +4.1%)	3.61×	2.77×
vtr_lu64peeng	2.45×	(-8.8%, +7.7%)	3.53×	2.74×
vtr_mcml	2.25×	(-4.0%, +5.8%)	7.17×	5.23×
Average	2.22×	(-5.9%, +6.7%)	5.01×	4.36×

*Like Tab. 1, BE (2-thread) integrated router has 8 runs, and others 3 runs.

Result Example (Dense Parallel)

threads	time/s	ratio	This is running on corescore_500_unrouted
1	128.3		Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
2	69.3	1.851371	
4	43.0	1.611628	
8	35.2	1.220204	
16	29.7	1.184936	
32	33.0	0.901212	
64	34.7	0.952381	



评分标准

- Final Project
  - 20% of the total grade
- You will get at most 80% of the final project socre if you only implement
  - Basic Parallel A* *or* Intuition Parallel
- You will get 90% of the final project score if you implement
  - Hash Distributed A* *or* Parallel Bidirectional A* *or* Dense Parallel
- You will get 100% of the final project score if you
  - create some novel methods and get better performance compared with our experiment results
- You should submit your **source code** and **experiment report**
  - Your report should include :
    - Brief introduction of your implementation
    - Experiment results (time acceleration and critical path cost)

附录：问题简化

- 1. Alt Source 化简
  – 每个 net 实际可能有 source 和 alternative_source，其中一个 source 建立连接即可完成 routing
  – 我们删掉了需要使用 alt_source 的 connection，每个 connection 只有一个 source 到 一个 sink
- 2. Connection 化简
  – Indirect Connection：source -> source_int -> sink_int -> sink
  – Direct Connection: source -> sink
  – 本次作业只需要考虑 indirect connection 中 source_int 到 sink_int 的计算

附录：START UP

- 从北大网盘下载压缩包
  - https://disk.pku.edu.cn/link/AAD923CB8CC59A48BFA9886F11A026AB9E
- 解压
  - tar -zxvf fpga-route.tar.gz
- 环境配置
  - 根据 README 配置环境
  - 你可以选择在自己的机器 或 我们提供的AMD服务器配置环境并编程
- Benchmark
  - 你可以在较小的benchmark上验证算法的初步效果（如 boom_med_pb，约1分钟内布线完成）
  - 之后在较大的benchmark进行验证（有些用例可能花费 1000-2000 s）