階層的逐次計算の Tascell言語による並列化/ HOPE言語による耐障害性

八杉 昌宏 (九州工業大学)

(平石 拓(京都大学)らとの共同研究)

M. Yasugi's Backgrounds

Developed programming language systems

- ABCL/EM4 [ICS 1992, PACT 1994]
  - Implementing ABCL (concurrent OO language) for a data-driven parallel computer
- OPA [PACT 1998, PDSIA 1999, ISHPC 2003]
  - Multiple threads over passive, adaptive objects
  - Synchronization and exception handing using dynamic scope
  - Lazy Task Creation with lazy frame conversion
- Tascell [PPoPP 2009, P2S2 2016]
  - "logical thread"-free efficient work-stealing framework
  - Backtracking-based load balancing with on-demand concurrency
    - Only when requested, each worker spawns a real task by temporarily backtracking and restoring its oldest task-spawnable state.
    - Using mechanisms for legitimate execution stack access
- HOPE [ICPP 2019 @Kyoto, Japan] ← Tascell の後はこの話

ここでTascellの話

タスク並列言語Tascellの設計・実装・応用

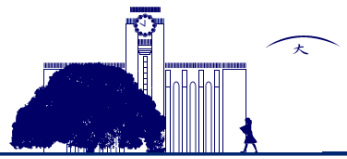
平石　拓
（学術情報メディアセンター　スーパーコンピューティング研究分野）

（狭義の・ここで扱う）タスク並列言語

- 多くの場合，既存の逐次言語（C，C++，Javaなど）の拡張
- 並列に実行可能な処理をタスクとして生成
  - 通常は計算資源（コア）数より非常に大きな数のタスクを同時に生成
  → タスク生成のコスト・オーバーヘッドの削減が処理系実装上重要
- タスクの同期ポイントを指定
- どのタスクをどのコアが実行するかは，言語処理系が自動的に，かつ動的に決定する

OpenMP Task, Cilk (Plus), Intel TBB*, X10, Chapel, MultiLisp, Massive Threads*, Tascell

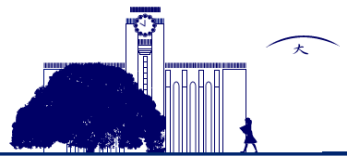
- *は言語ではなくライブラリ



Cilk

- おそらく最も有名なタスク並列言語
  - Frigo, M., Leiserson, C. E. and Randall, K. H.: *The Implementation of the Cilk-5 Multithreaded Language*, Proceedings of the ACM SIGPLAN 1998
- もともとMITで開発（Cilk, MIT Cilk）
  → ベンチャー企業立ち上げ（Cilk++）
  → Intelが買収し，現在はIntel C++ Compilerの一部（Cilk Plus）
- Cilk Plusの仕様自体はオープン，現在はGCCでも利用可能

- Lazy Task Creation (LTC)という実装手法により，タスク生成にかかるコストを（例えばOSのスレッド生成と比べて）大幅に削減



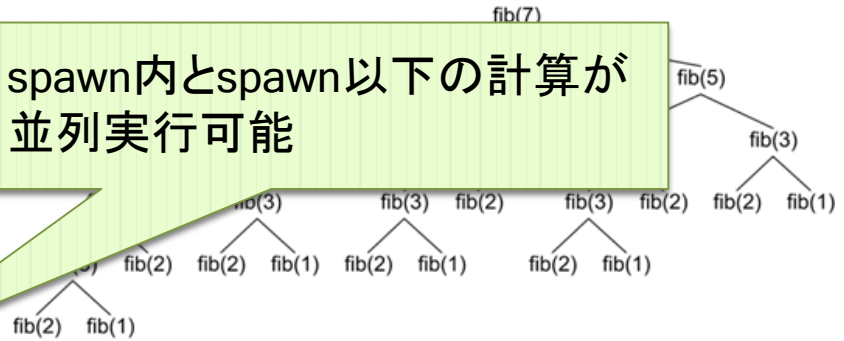
Cilkのプログラム例

- Fibonacci using a doubly recursive algorithm
  - タスク並列言語のベンチマークプログラムの定番

  ※このプログラム自体に実用的な意味はない（明らかに効率が悪い）

```
cilk int fib(int n)
{
      if (n <= 2)
            return (1);
      else {
            int x, y;
            x = spawn fib(n - 1);
            y = spawn fib(n - 2);
            sync;
            return (x + y);
}}
```

spawn内とspawn以下の計算が並列実行可能

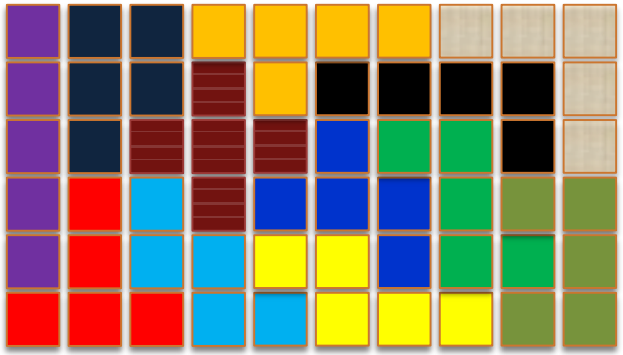


同一スコープ内の全てのspawnの終了を待ち合わせる

Fibの図： http://www.introprogramming.info/english-intro-csharp-book/read-online/chapter-10-recursion/

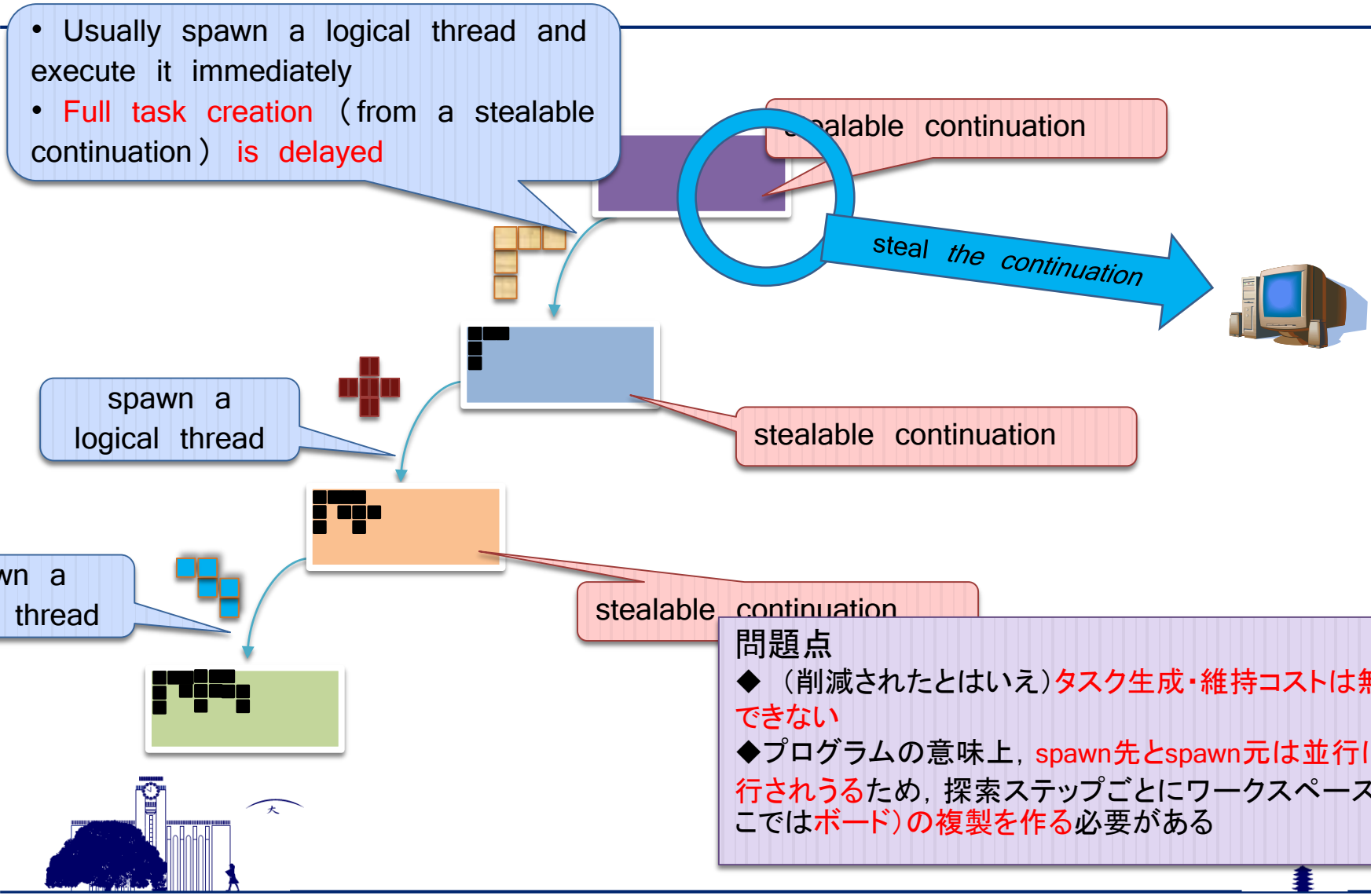
例：Pentomino

- Pentomino Puzzle全解探索
  - Pentomino: a block consists of five attached squares
  - Fill a 6×10 rectangular board with the 12 different pentominos



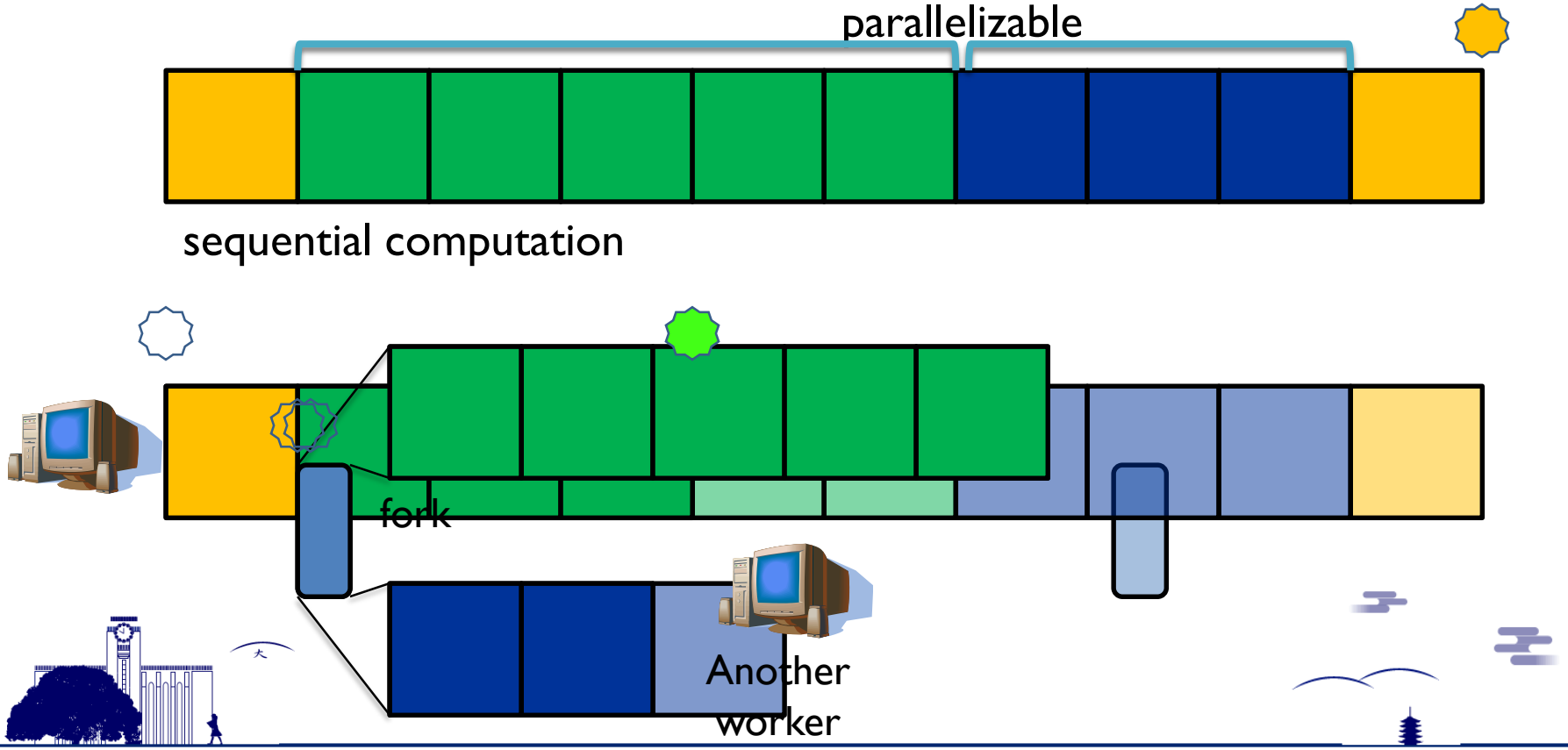
- 並列バックトラック探索で解きたい
  - データ並列処理として記述することが困難で，
    タスク並列言語が得意な例
- 考え方の大部分は，実用的な探索アルゴリズムにも適用可能

Cilkプログラムの動作と問題点

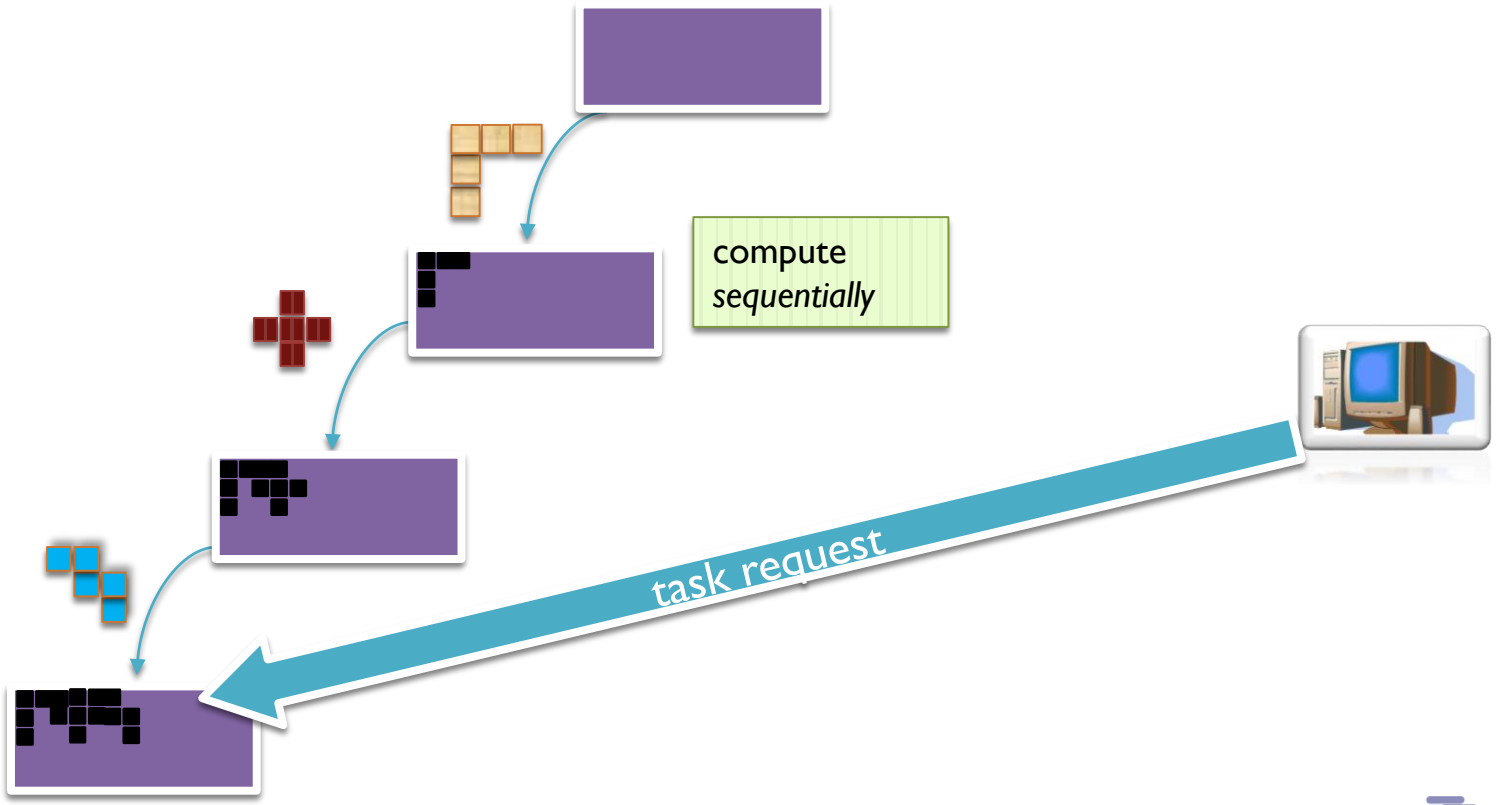


タスク並列言語Tascell [PPoPP ’09]

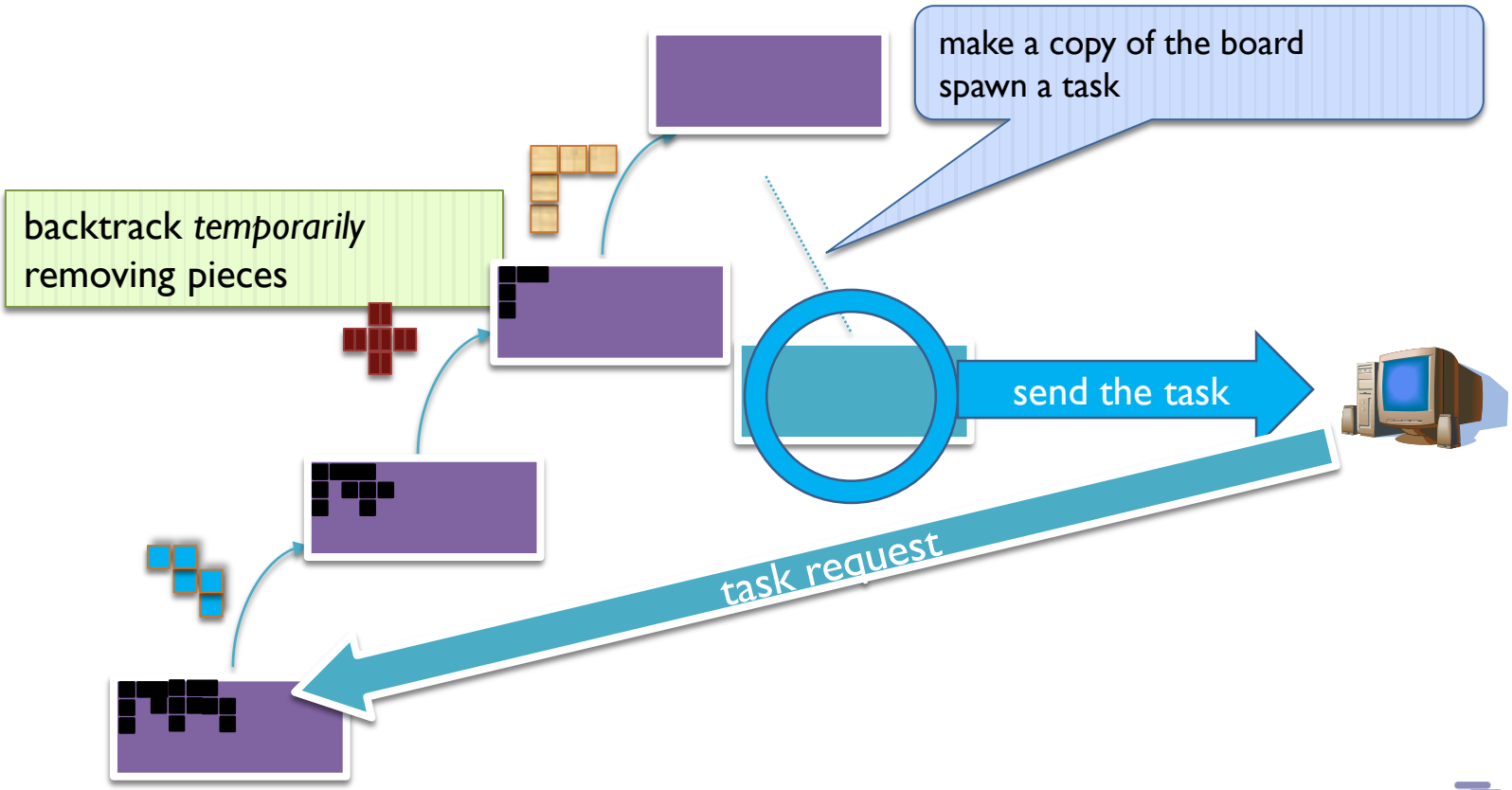
- Backtracking-based load balancing approach
- 並列性の発生そのものを，必要時まで遅らせる



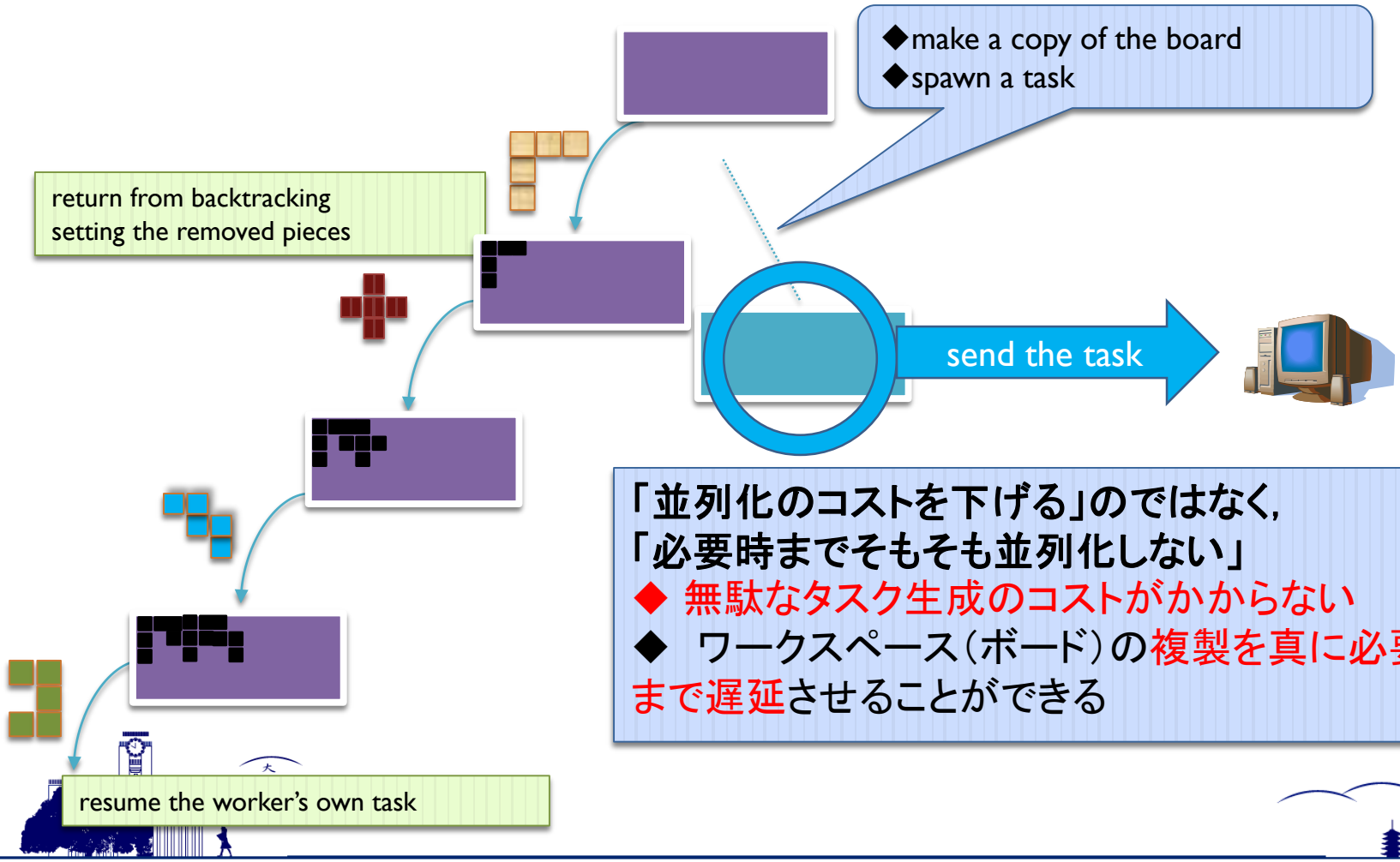
Tascellプログラムの動作



Tascellプログラムの動作



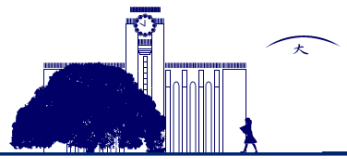
Tascellプログラムの動作



「並列化のコストを下げる」のではなく,
「必要時までそもそも並列化しない」
◆ 無駄なタスク生成のコストがかからない
◆ ワークスペース(ボード)の複製を真に必要になるまで遅延させることができる

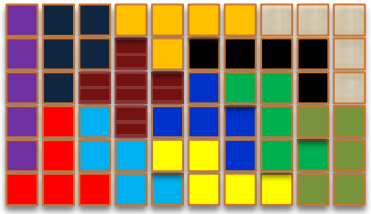
Tascell言語

- Cの拡張並列言語
- 並列for構文
- 一時的バックトラック
  - ワーカはタスク要求がない限り(spawnせず)逐次実行
  - 他のワーカからタスク要求を受けると
    - 最古の並列forまで一時的にバックトラックし,
    - タスクを生成した後
    - 一時的バックトラックから復帰して自らの計算を再開する
  - 「一時的バックトラック」「一時的バックトラックからの復帰」時の処理を記述するための, dynamic_wind構文を備える(c.f., Scheme)



Pentomino全解探索 in Tascell

```
worker int search (k, j0, j1, j2, task pentomino *tsk)
{
  for (int p : j1, j2)  {   /* 並列for:ピースの種類 */
    for (d=0 ; d<4 ; d++) { /* 逐次for:ピースの方向 */
      dynamic_wind {
        ピース p を方向 d で置く
      } {
        if (解が見つかった？)  s++;
        else   s += search(…);  // recursive call
      } {
        ピース p を取り除く
      }
    }
  } handles pentomino (int i1, int i2) {  /* i1--i2 is a subrange of j1--j2 */
     /* executed to initialize :in fields of a spawned task */
    { copy_piece_info (this.a, tsk->a); copy_board (this.b, tsk->b); this.k=k; this.i0=j0; this.i1=i1; this.i2=i2; }
       /* executed to get the result by referring  :out fields */
     s+=this.s;
  }
  return s;  }
task_exec pentomino { return search(…); }
```



Background

High productivity, scalability, load balancing, and fault tolerance

to develop high performance and robust applications including irregular ones

for massively and extremely parallel computing systems.

Work-stealing frameworks (Lazy Task Creation, Cilk, Tascell, …)

- provide good load balancing for many parallel applications, including irregular ones written in a divide-and-conquer style
- However, work-stealing frameworks enhanced by fault-tolerant features (such as checkpointing) do not always work well

Contributions

We propose a new ``hierarchical work omission''-based parallel execution model called HOPE.

- HOPE programmers' task is to specify
  - which regions in imperative code can be executed in sequential but arbitrary order and
  - how their partial results can be accessed.
- Every worker has the entire work of the sequential program with its own planned execution order (SPMO; single program multiple order).
- Workers (and runtime systems) automatically exchange partial results to omit hierarchical subcomputations.

We discuss how to implement this new idea as an efficient, scalable and fault-tolerant dynamic load balancing programming/execution framework

- Fault tolerance without a single point of failure by assuming fail-stops
- the language, compiler, and runtime system.

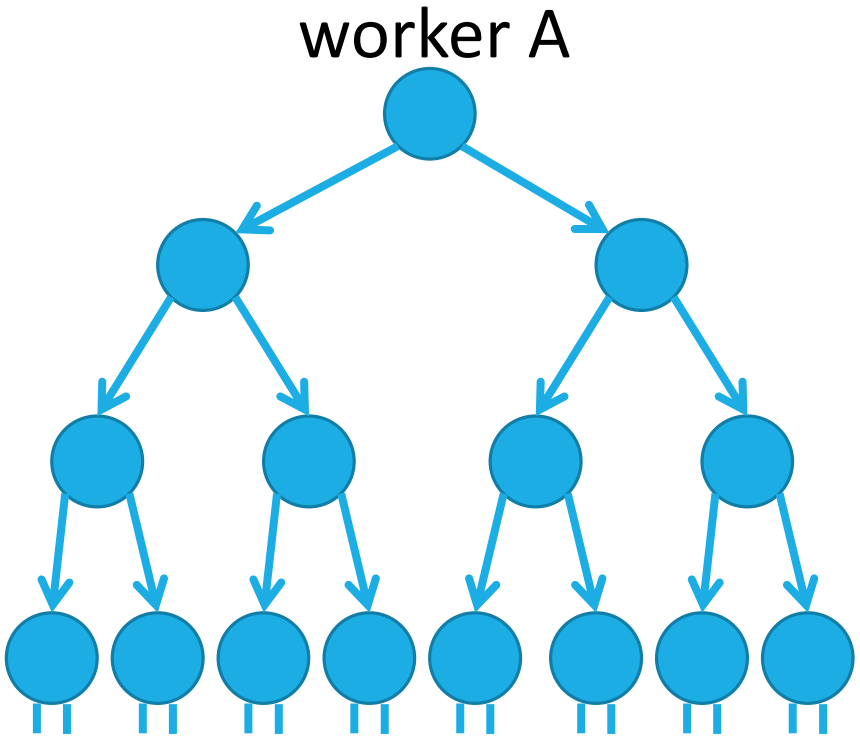
“HOPE” is NOT what you think

A HOPE worker does

- NOT have only an assigned (fixed) part of the entire work
- NOT wait for slow workers
- NOT have a fixed communication pattern
- NOT spawn a task/thread
- NOT steal a task/thread
- NOT wait for a result
- NOT broadcast a result
- NOT recover/restart (a stopped part)
- NOT save/restore/rollback an execution state

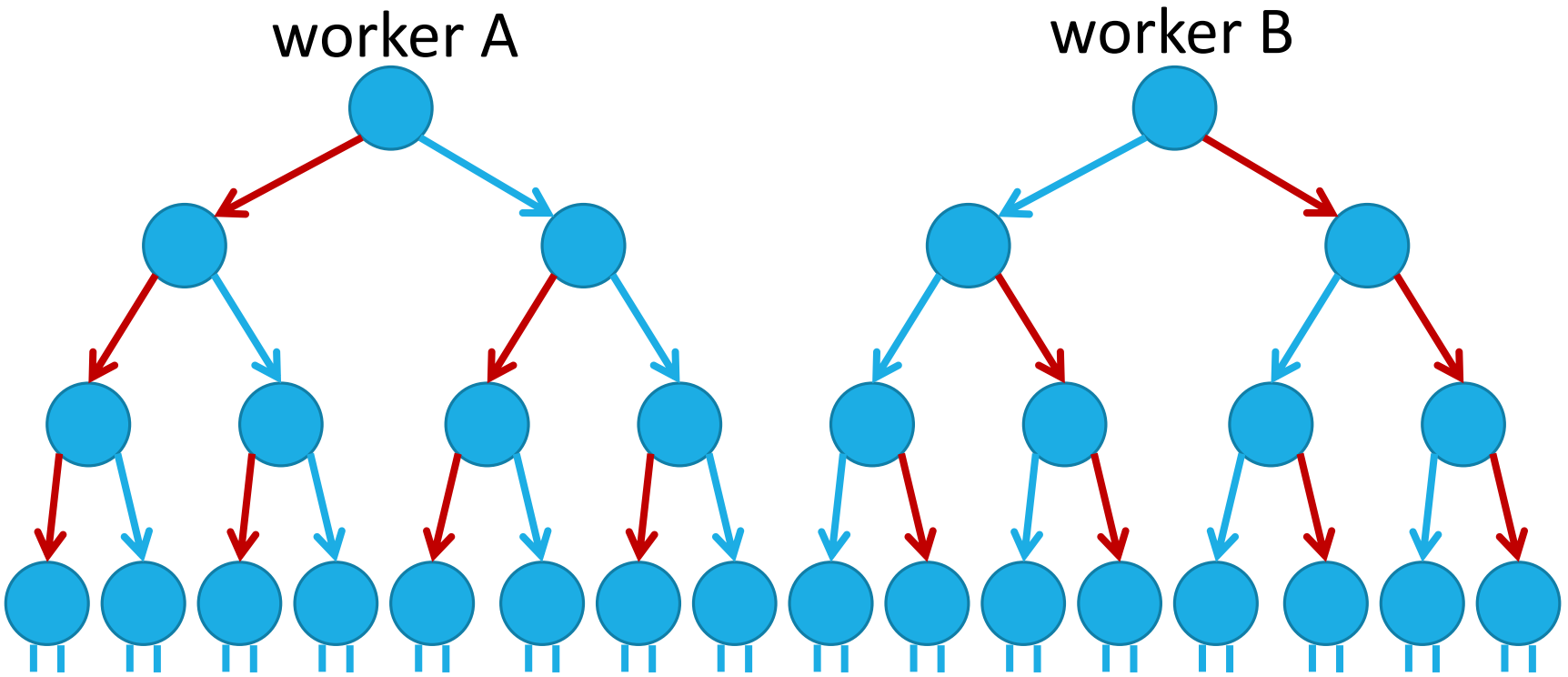
Basic Idea (1/3)

A worker performs a divide-and-conquer computation sequentially.



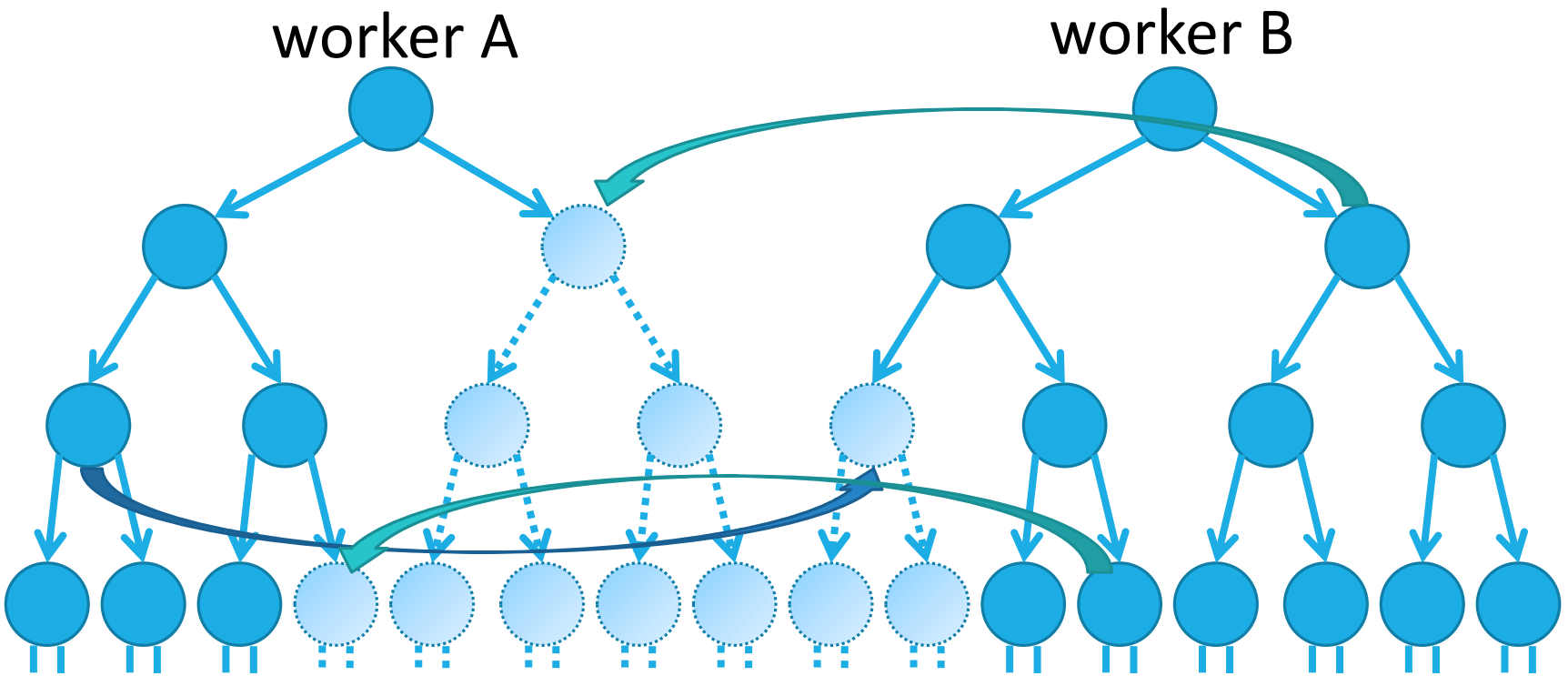
Basic Idea (2/3; full redundancy for fault tolerance)

Every worker performs the same program with its own planned execution order (SPMO; single program multiple order).



Basic Idea (3/3; parallel speedups)

Exchange partial results for hierarchical omission



Our Approach

The HOPE Language: Directives:

- Arbitrary execution order (≠ spawning concurrent tasks/threads)
- How to access to partial results

Pre-shared plans:

- Every worker has its own planned execution order of a hierarchical computation

Variable-length addresses to identify hierarchical subcomputations

Compiler (translator)

- into C with nested functions (L-closures [Yasugi+, CC 2006])
- We solve dilemmas: 3 execution modes and dynamic execution mode switching

Runtime: Message Mediation Systems (MMSs) (distributed and federated)

- as a local storage and an automatic exchanger of messages of partial results

Outline

Motivating Example

The HOPE language

Our Approach

Implementation

- Message Mediation Systems
- Pre-shared Plans
- The Compiler

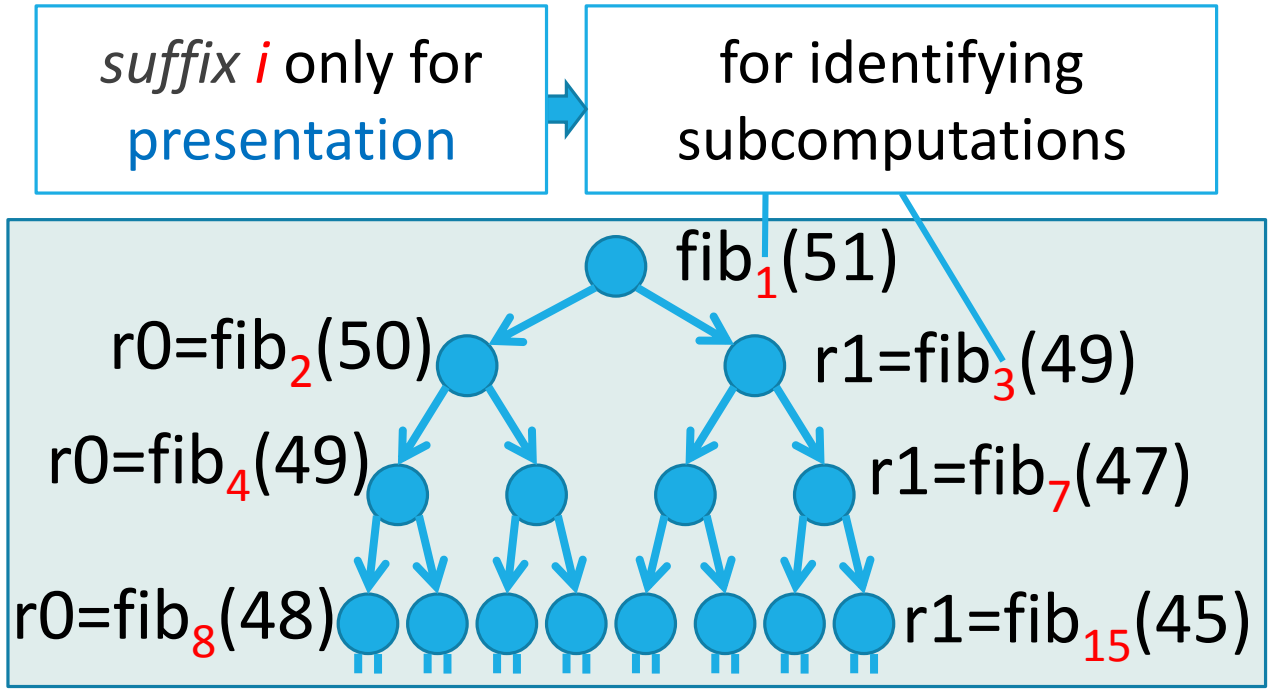
Evaluation

Discussion

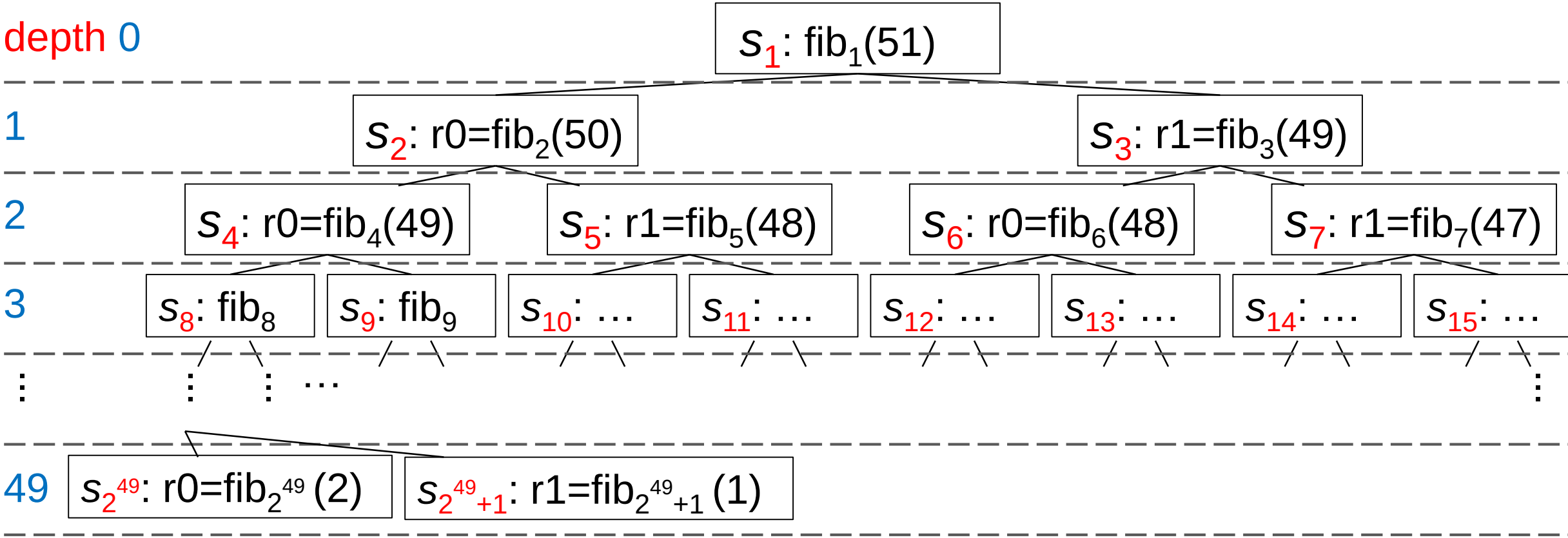
Motivating Example

A doubly-recursive computation for Fibonacci
(typically employed as an irregular example; NOT a fast algorithm)

```
int fib_i (int n) {
    if (n <= 2) return 1;
    int r0, r1;
    r0 = fib_2i (n - 1);
    r1 = fib_2i+1(n - 2);
    return r0 + r1;
}
```



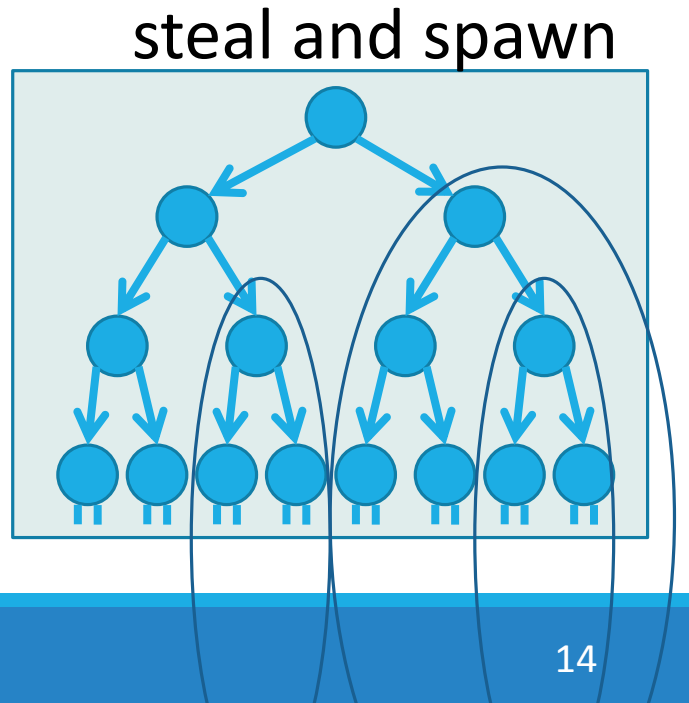
Hierarchical subcomputations in $fib_1(51)$ and their depths

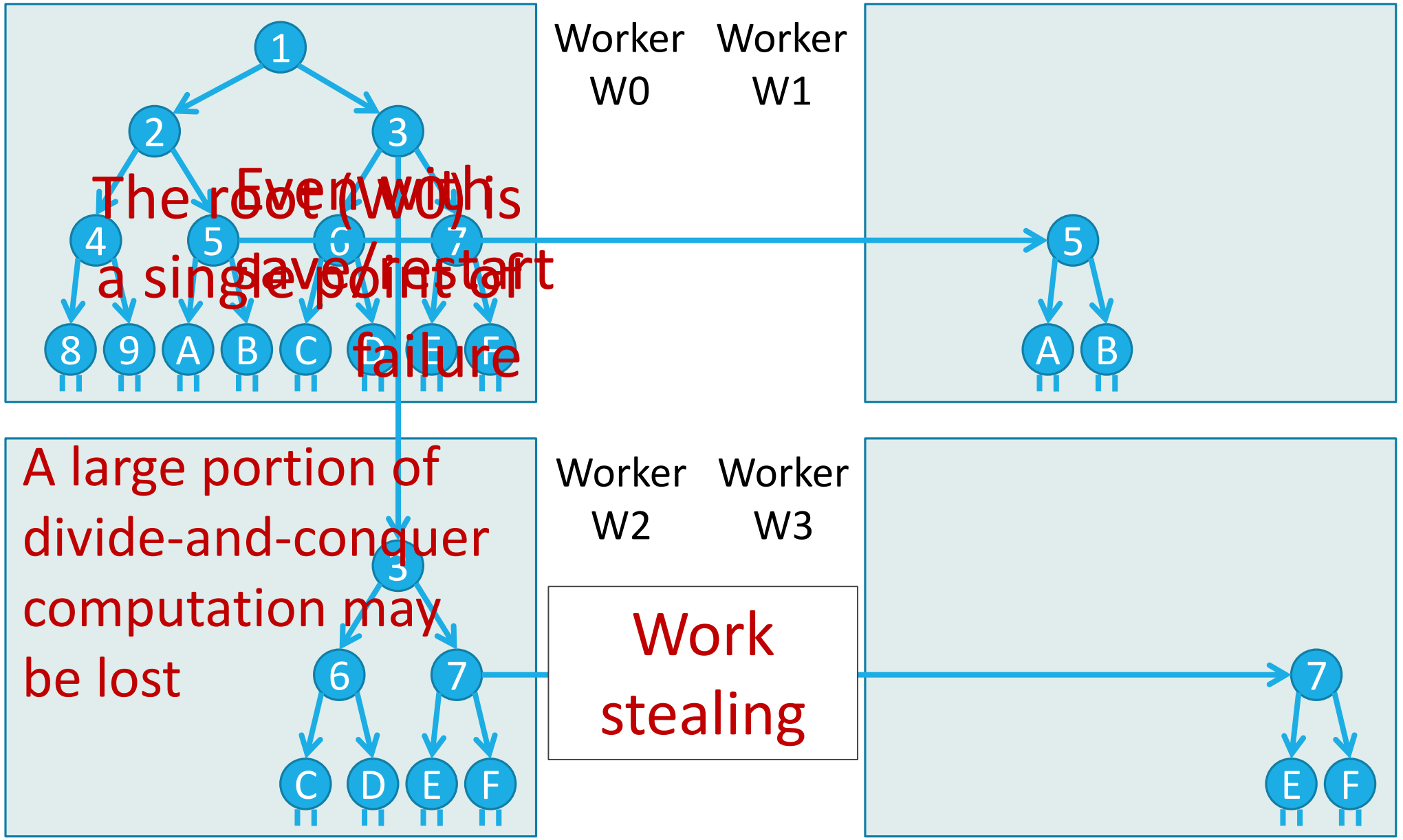


An existing work-stealing framework: Tascell [Hiraishi,Yasugi+, PPoPP 2009]

(On demand) concurrency (between thief and victim)

```
task tfib { in: int n; out: int r };
worker int fib (int n) {
    if (n <= 2) return 1;
    int r0, r1;
    do_two                    // Tascell's construct
      r0 = fib(n - 1);
      r1 = fib(n - 2);        // skip if a task tfib is spawned
    handles tfib {
      { this.n = n - 2; }  //  for spawning a task tfib
      { r1 = this.r; }     //  merge the result of tfib
    }
    return r0 + r1;
}
```




Worker W0
Worker W1
Worker W2
Worker W3
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
Even with save/restart
The root (W0) is a single point of failure
A large portion of divide-and-conquer computation may be lost
Work stealing

The HOPE Language

w/o concurrency :
Subcomputations may (re)use a single workspace

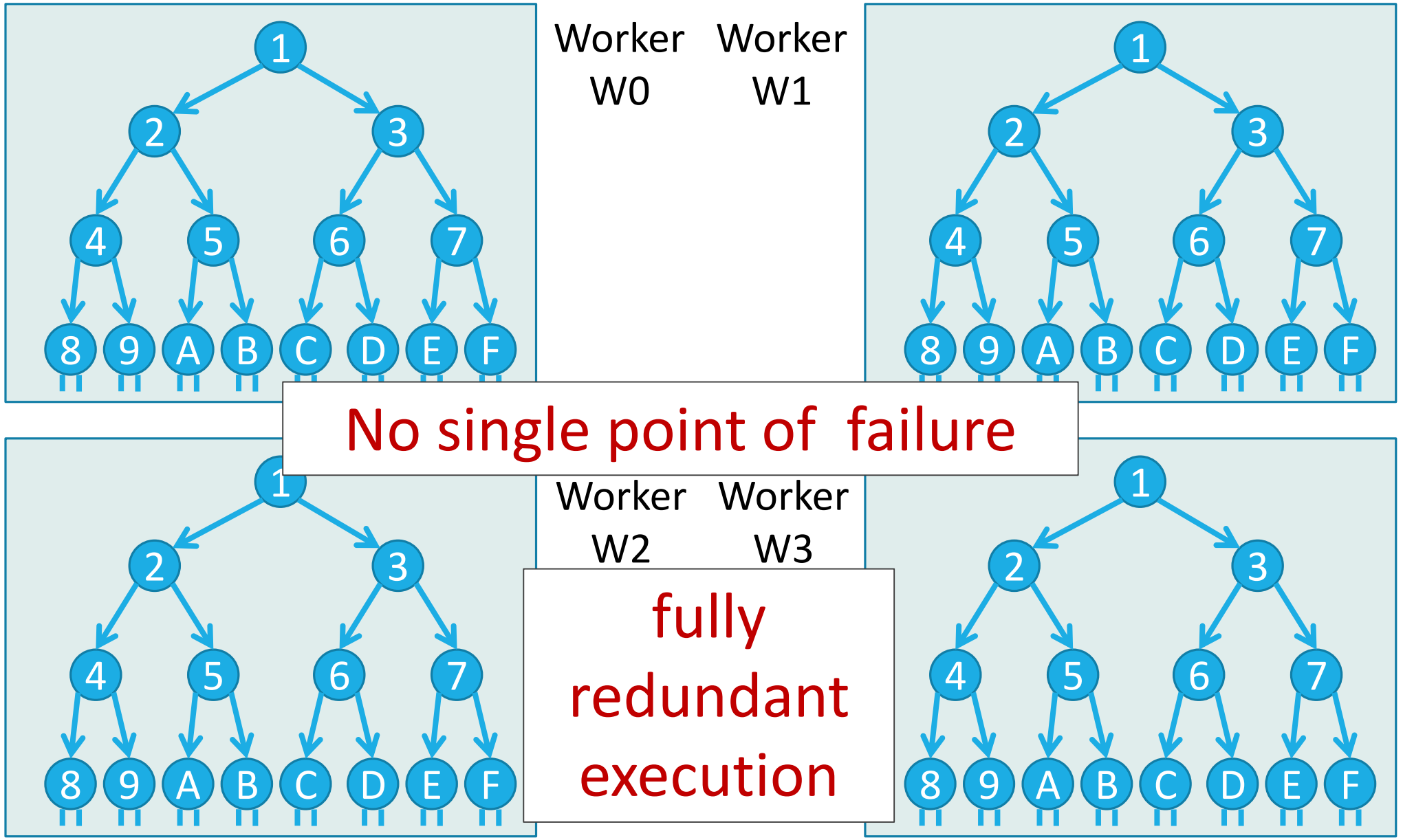
The HOPE Language

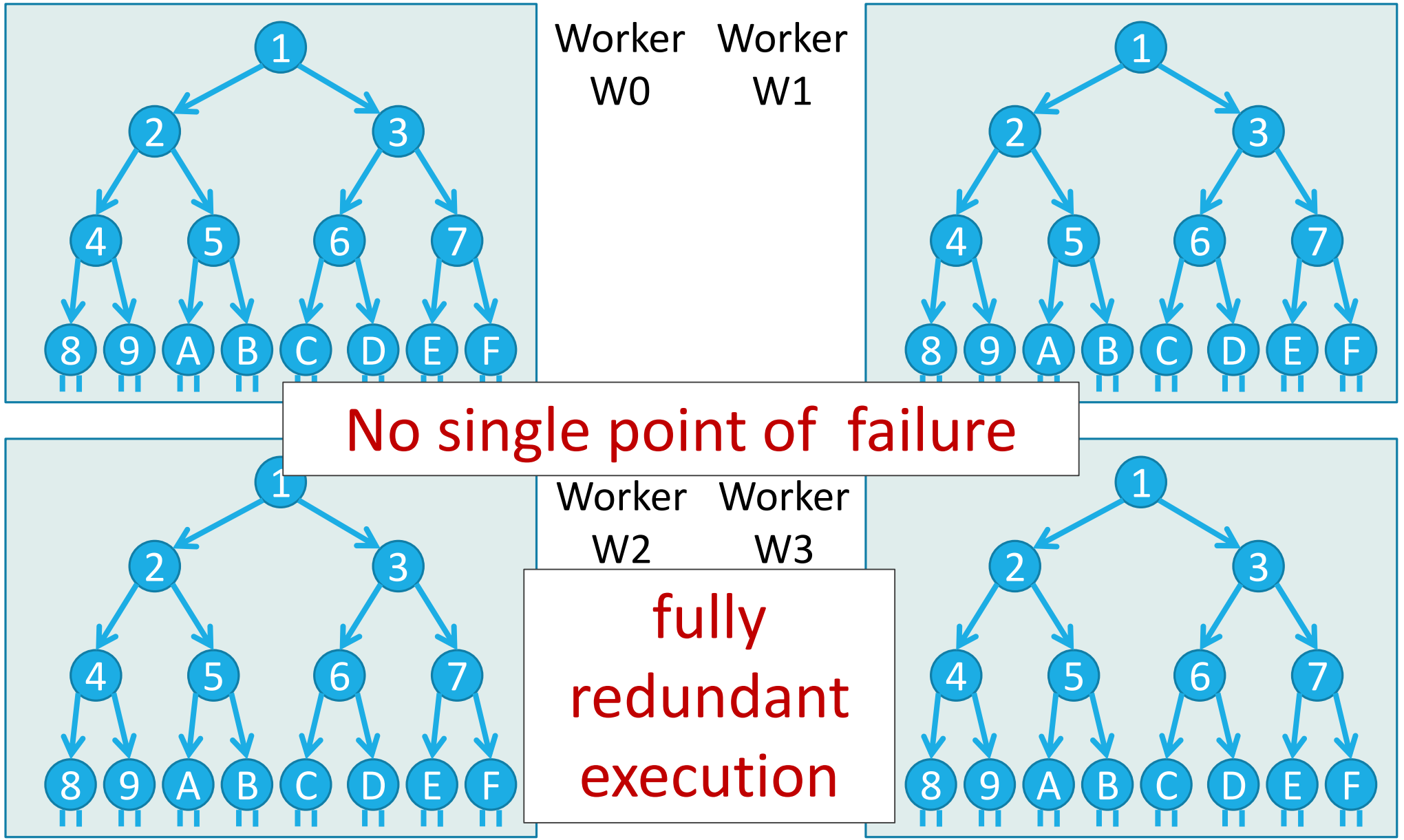
Directives

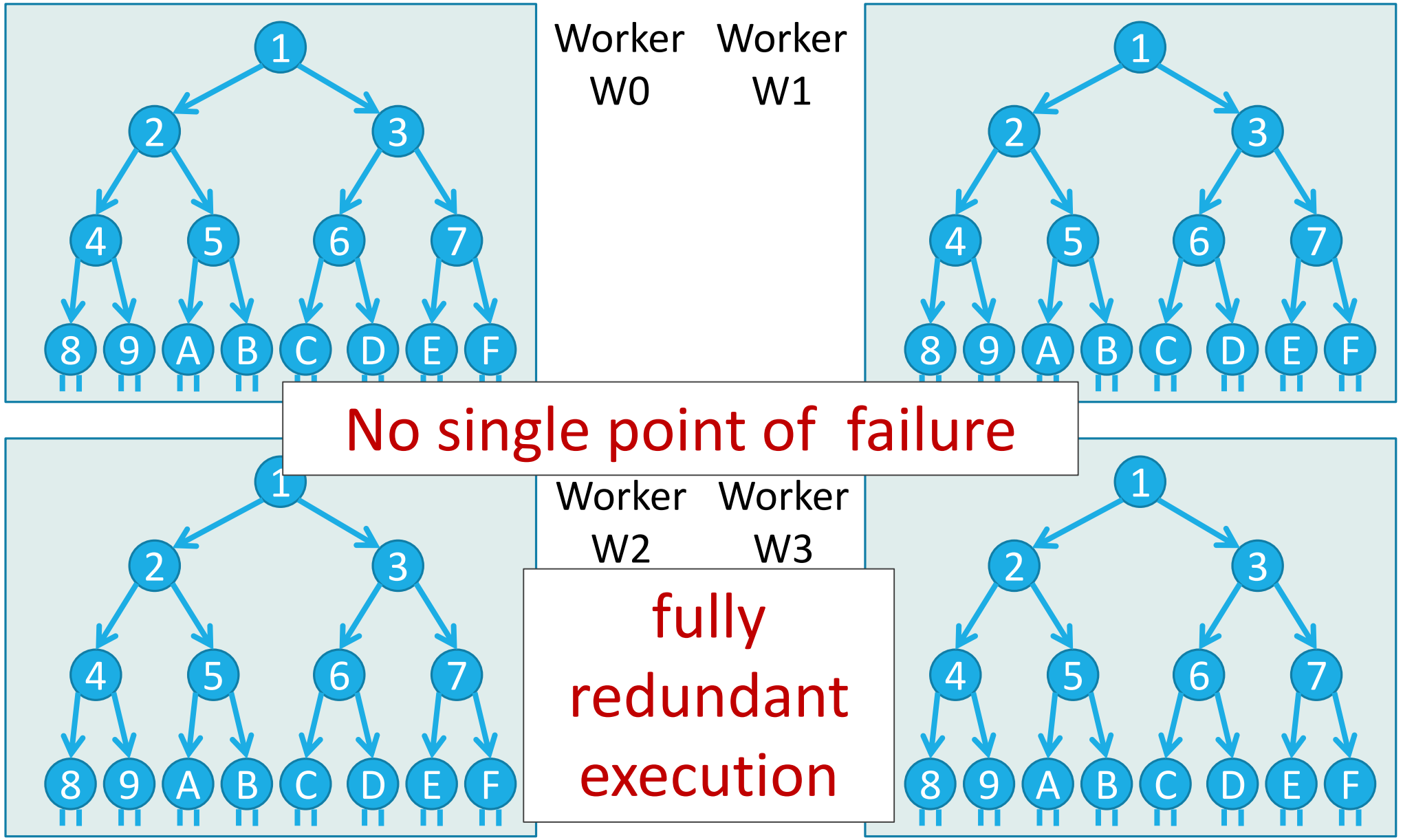
```
int fib (int n) {
    if (n <= 2) return 1;
    int r0, r1;
    #pragma hope do_two  // arbitrary order
    #pragma hope omissible result(r0)
    r0 = fib(n - 1); // may exchange partial result r0 for omission
    #pragma hope omissible result(r1)
    r1 = fib(n - 2); // may exchange partial result r1 for omission
    return r0 + r1;
}
```

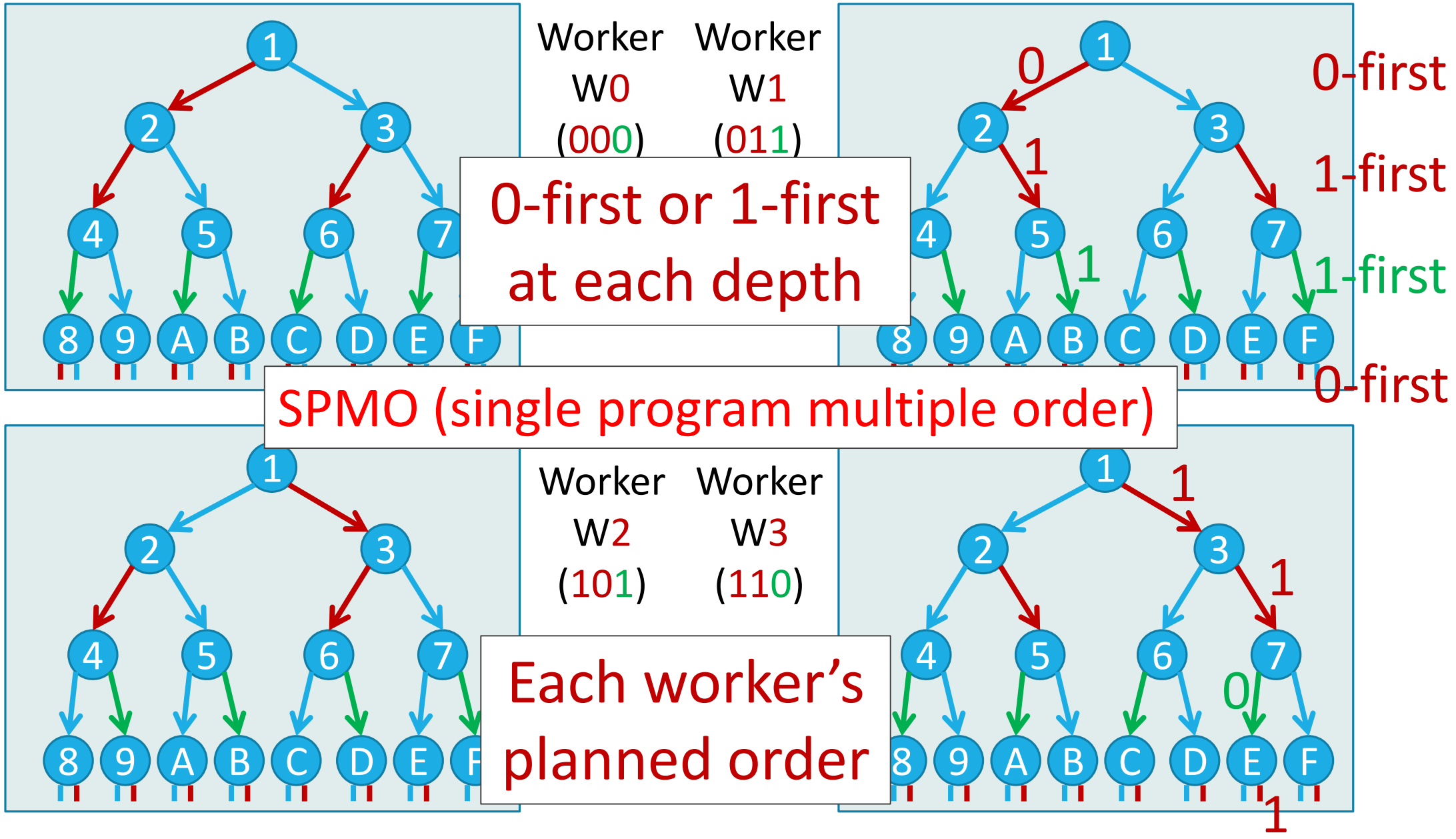
- Results are not always on the left-hand side.

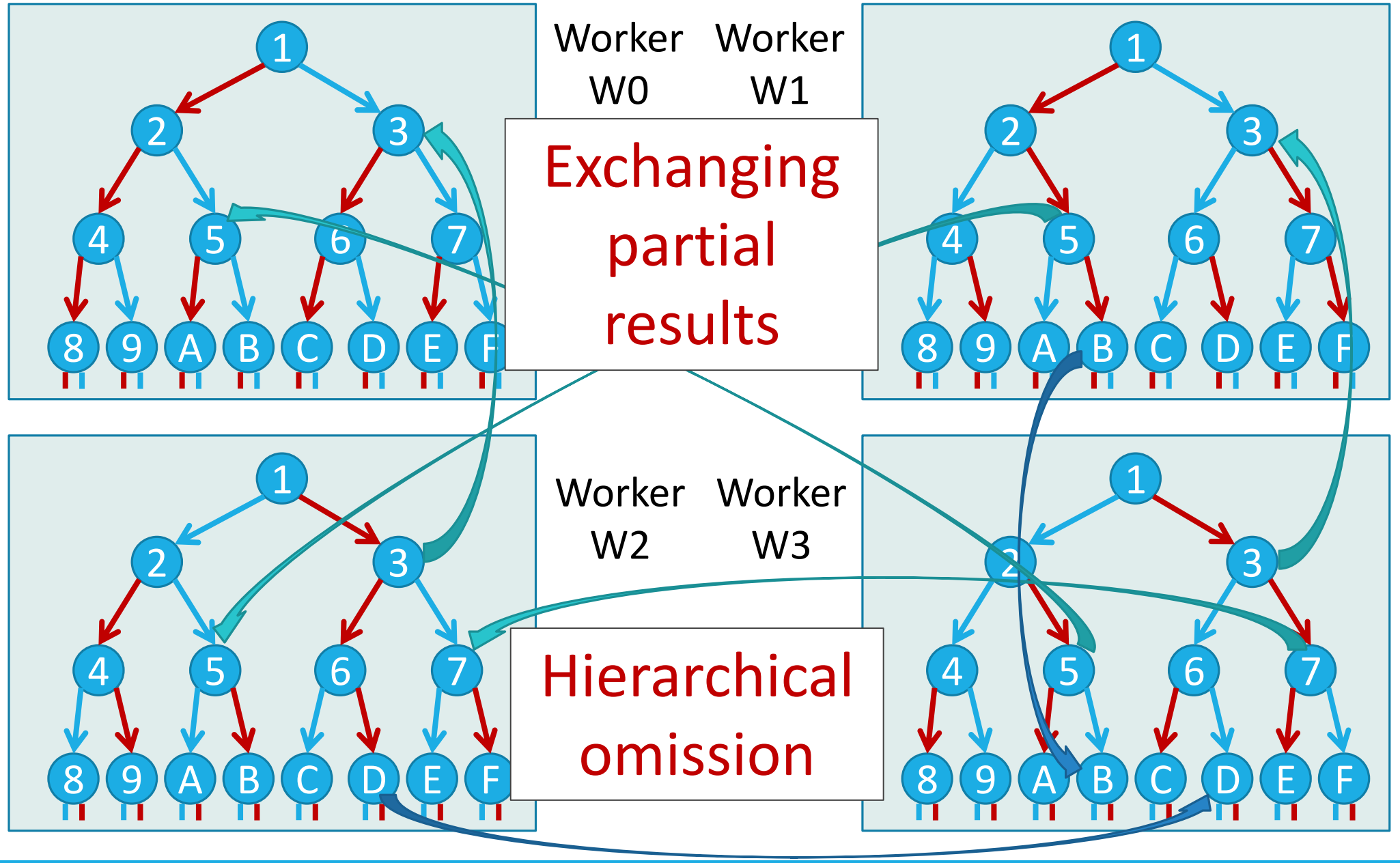
Our Approach


Worker W0
Worker W1
Worker W2
Worker W3
No single point of failure
fully redundant execution


Worker W0
Worker W1
Worker W2
Worker W3
No single point of failure
fully redundant execution

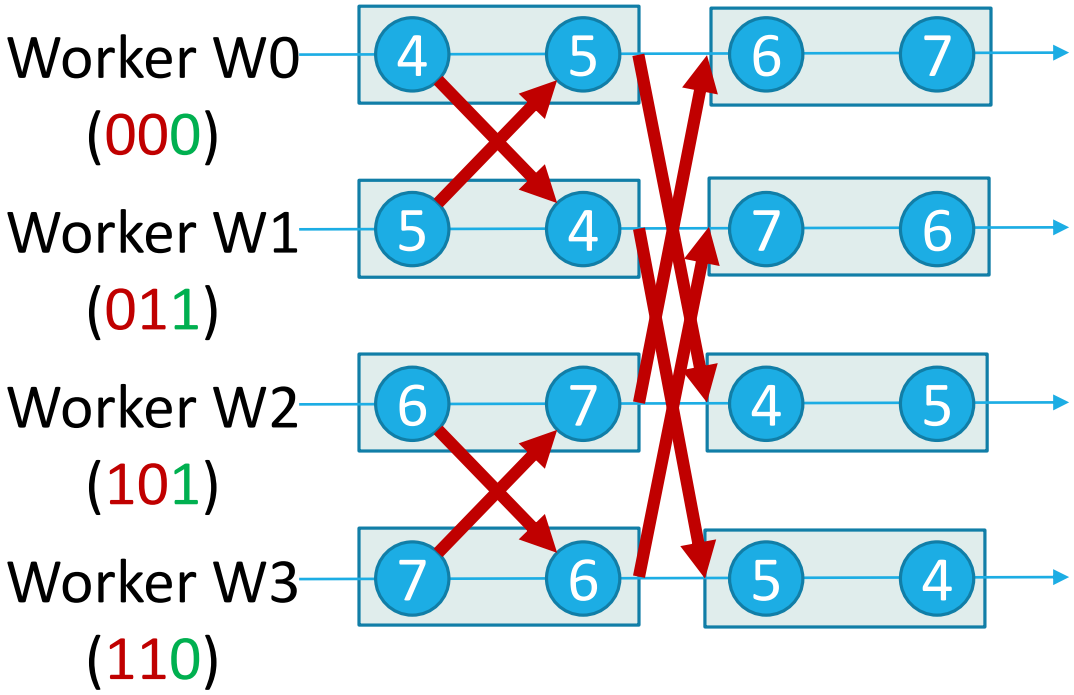

Worker W0
Worker W1
Worker W2
Worker W3
No single point of failure
fully redundant execution

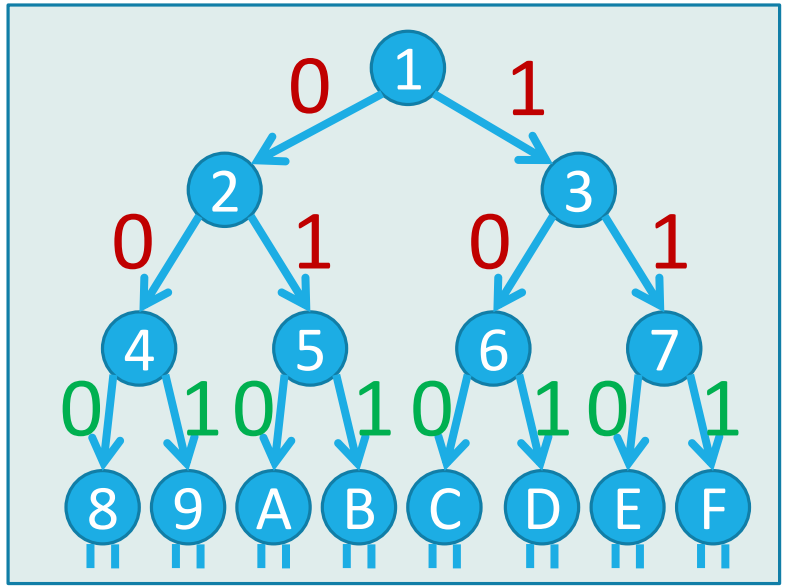

Worker W0 (000)
Worker W1 (011)
Worker W2 (101)
Worker W3 (110)
0-first or 1-first at each depth
SPMO (single program multiple order)
Each worker's planned order
0-first
1-first
1-first
0-first


Worker W0
Worker W1
Exchanging partial results
Worker W2
Worker W3
Hierarchical omission
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

Another example of hierarchical omission

When ignoring depth 3 and more





Implementation

The HOPE Compiler

Translation of a HOPE program into a C program with MMS API calls

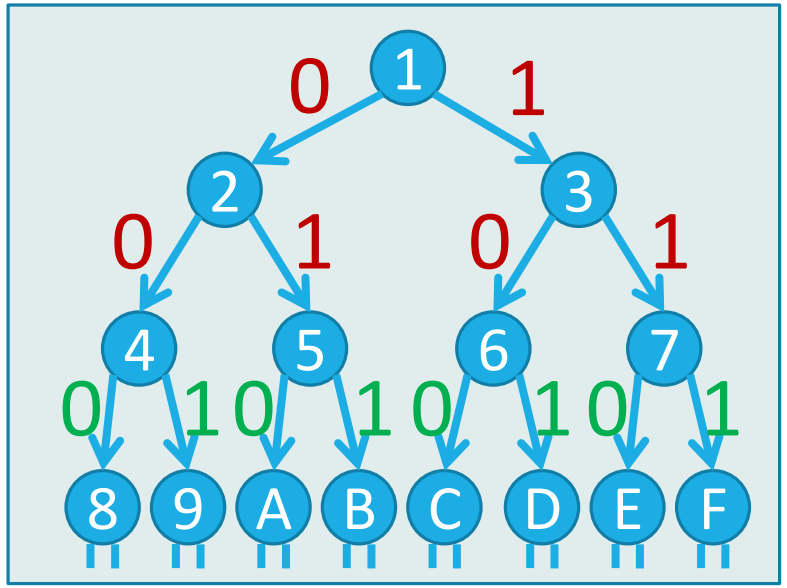
Naive translation would involve the following issues to be addressed:

Dilemma 1 (collaboration cost):

- ☺ Hierarchical omission of subcomputations
- ☹ Overheads of writing/checking partial results
- ☹ Costs of maintaining the “current” VL address

Dilemma 2 (multiple-order cost):

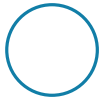
- ☺ SPMO reduces overlapping of subcomputations
- ☹ Costs of order selection
- ☹ Branch-predictor unfriendly



Dynamic execution mode switching with Nested functions

The “check” mode 

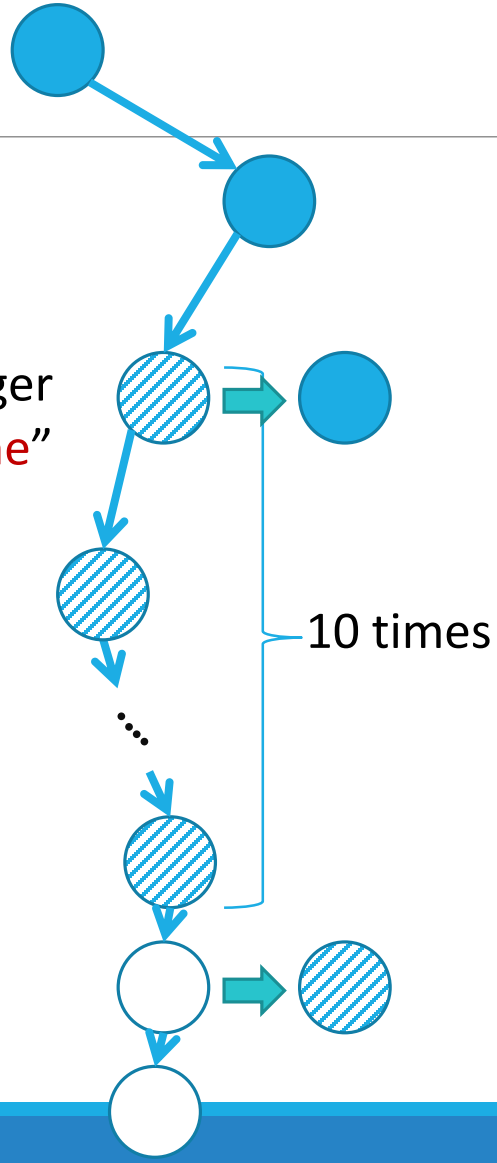
The “non-check” mode 

The “work-first” mode 

Dynamic execution mode switching (DEMS)

- Changes a past choice of modes

- Legitimate execution stack access [Yasugi+, CC 2006] to handle VL addresses and partial results



Evaluation

Environment/Benchmark

No cut-offs
(No thresholds)

The K computer (per node)

- SPARC64 VIIIfx 2GHz 8-core
- Tofu Interconnect
- FujitsuMPI（OpenMPI based）
- Fujitsu C/C++ Compiler 1.2.0
  - with -O2 optimizers
- Nested function
  - LW-SC [Hiraishi+ 2006]

Employing

- 1, 2, 4, 8, 16,…, 256, 512, 1024 nodes
- 7 workers/node

Benchmarks

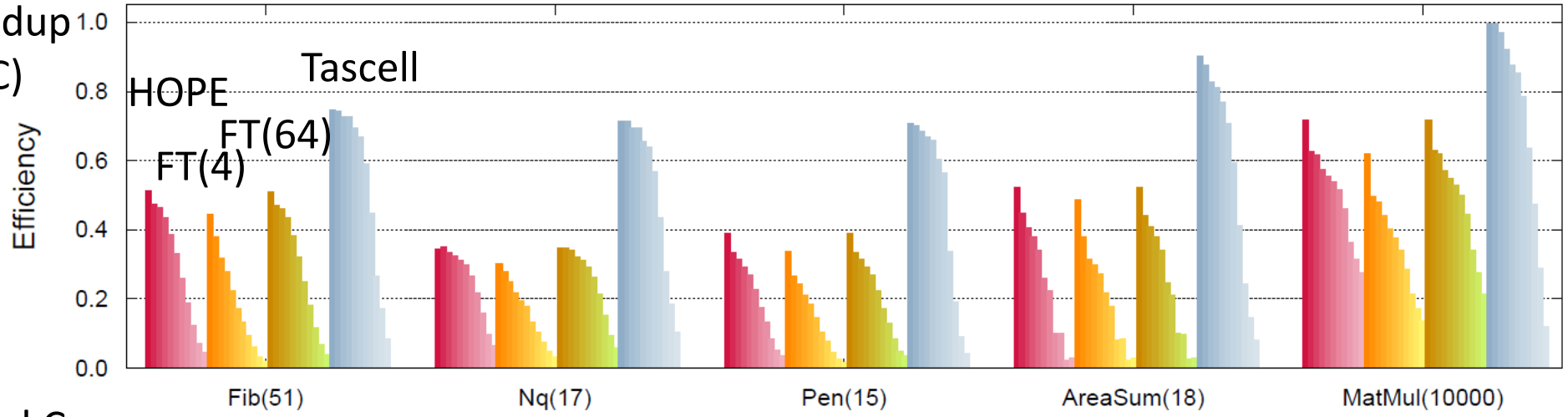
- Fib(*n*) Fibonacci
- Nq(*n*) *n*-queen problem
- Pen(*n*) Pentomino problem
- AreaSum(*n*) AreaSum
- MatMul(*n*) Matrix Multiplication

Fault injection

- FT(*n*) one out of *n* workers
- simulated by discarding incoming/ outgoing system messages

Compare HOPE and Tascell

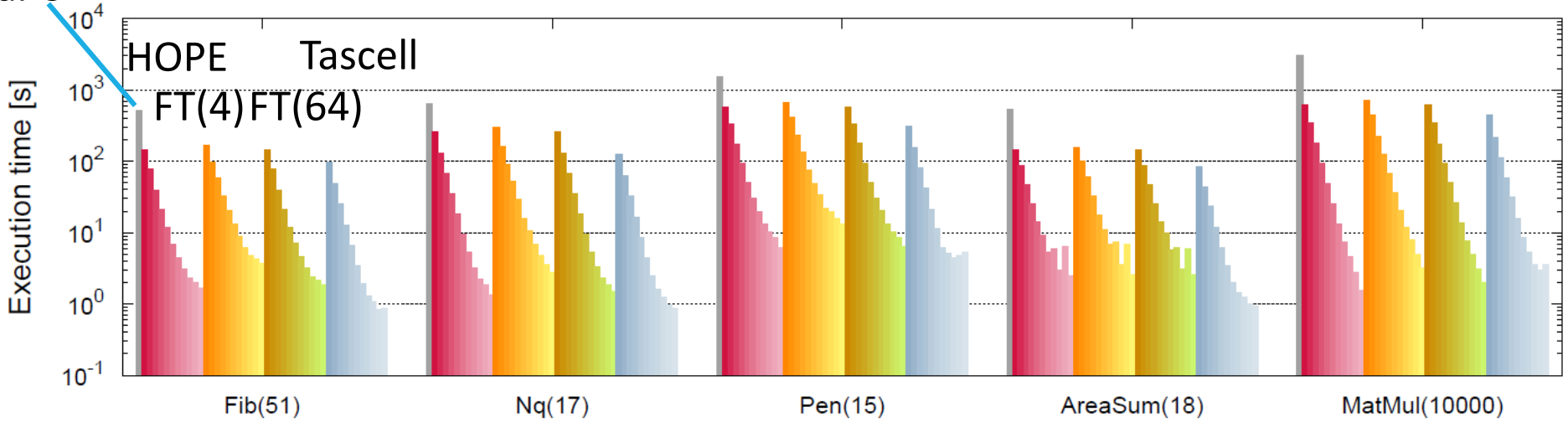
Ideal speedup
(to serial C)

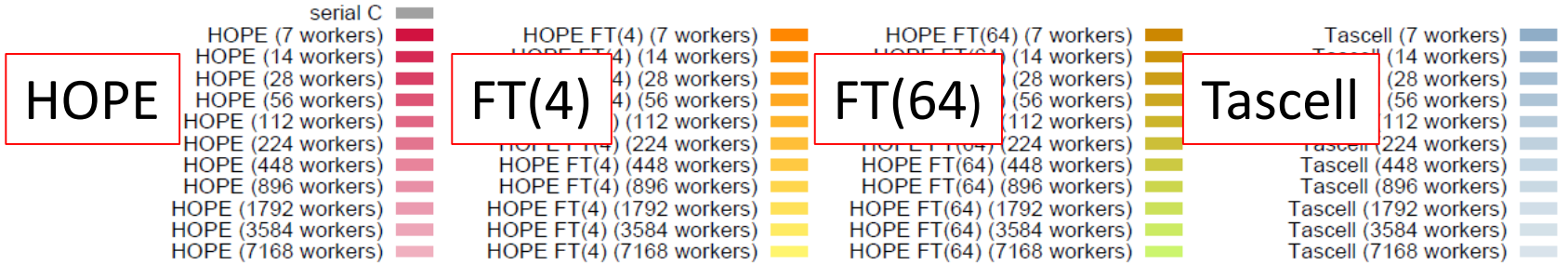


serial C

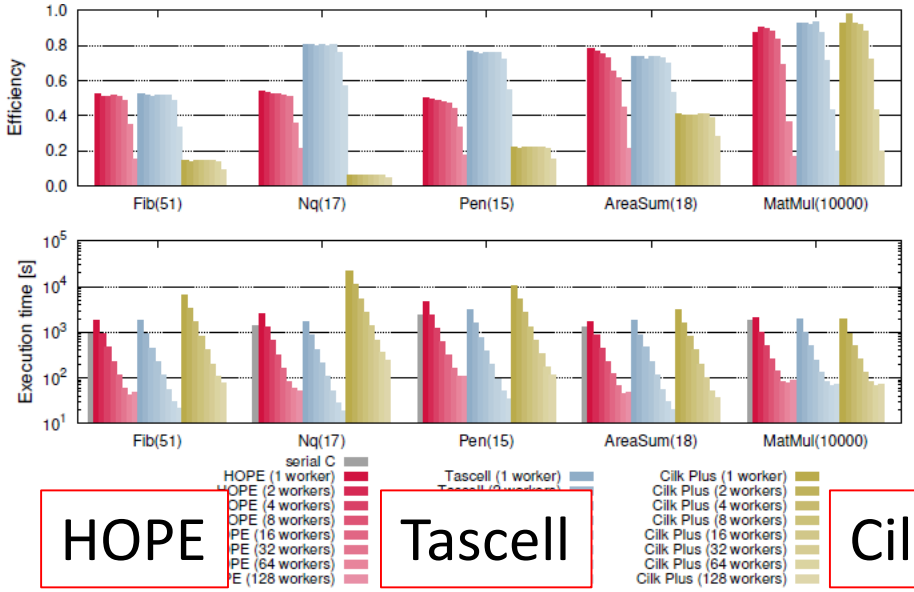
1000 s

1 second





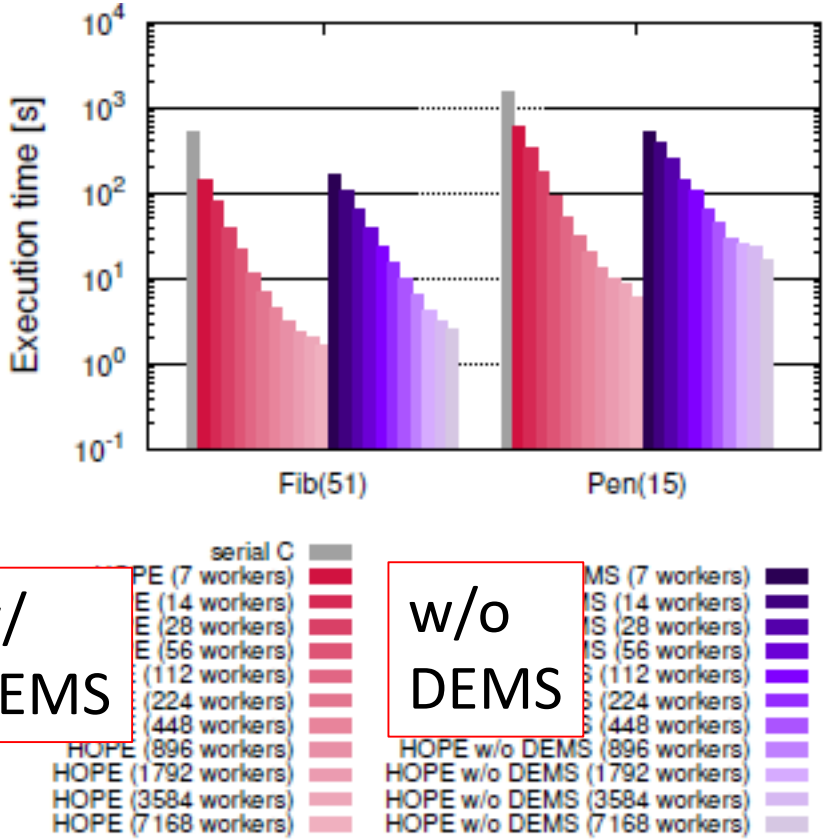
Other Evaluations



HOPE

Tascell

Cilk Plus



w/ DEMS

w/o DEMS

>75x faster execution

Implementations	execution modes				elapsed times [s] (and relative execution times)			
	check	non-check	work-first	DEMS	Fib(41)	Fib(51)	Pen(13)	Pen(15)
base + Sec. 6	on	off	off	off	433.3 (360.3)	-	907.8 (220.9)	-
w/ non-check	on	on	off	on	3.432 (2.853)	372.9 (3.256)	12.00 (2.919)	959.9 (3.110)
w/ work-first	on	on	on	on	1.203 (1.000)	114.5 (1.000)	4.109 (1.000)	308.7 (1.000)
w/o DEMS	on	on	on	off	1.228 (1.021)	122.5 (1.070)	13.01 (3.165)	432.2 (1.400)

>2.8x faster execution

Summary: What is “HOPE”?

Every HOPE worker (with its runtime system)

- has the entire work of a divide-and-conquer program to compute the result independently
- performs the entire hierarchical computation sequentially
- uses its own fixed/planned execution order of the hierarchical computation
- MAY omit a hierarchical subcomputation if the result is locally available (handles the result of a subcomputation as an eliminator of the subcomputation)
- MAY exchange the result of a subcomputation with only relevant workers
- CAN guess the current participants of a subcomputation based on local availability of partial results and the entire pre-shared plans