HOPE: A New Parallel Execution Model Based on Hierarchical Omission

Masahiro YASUGI
(Kyushu Institute of Technology, Japan)

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 [this presentation, To appear in ICPP 2019]

Background

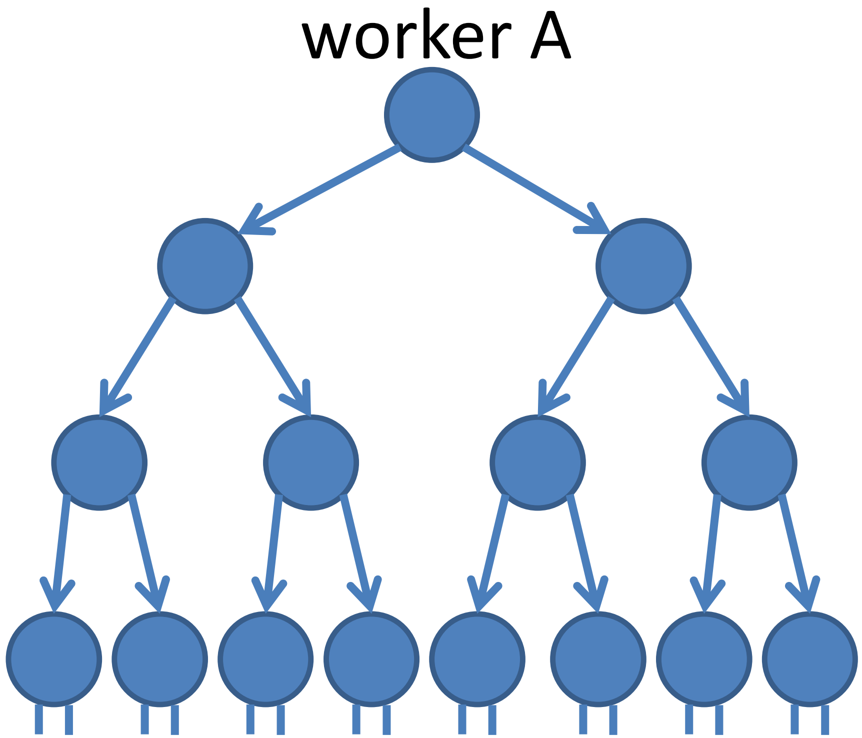
- High productivity, scalability, load balancing, and fault tolerance
- to develop high performance and robust applications including irregular ones
- for massively parallel computing systems.

Contributions

- We propose a new ``hierarchical work omission''-based parallel execution model called 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).
  - 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 without a single point of failure.
  - the language, compiler, and runtime system.

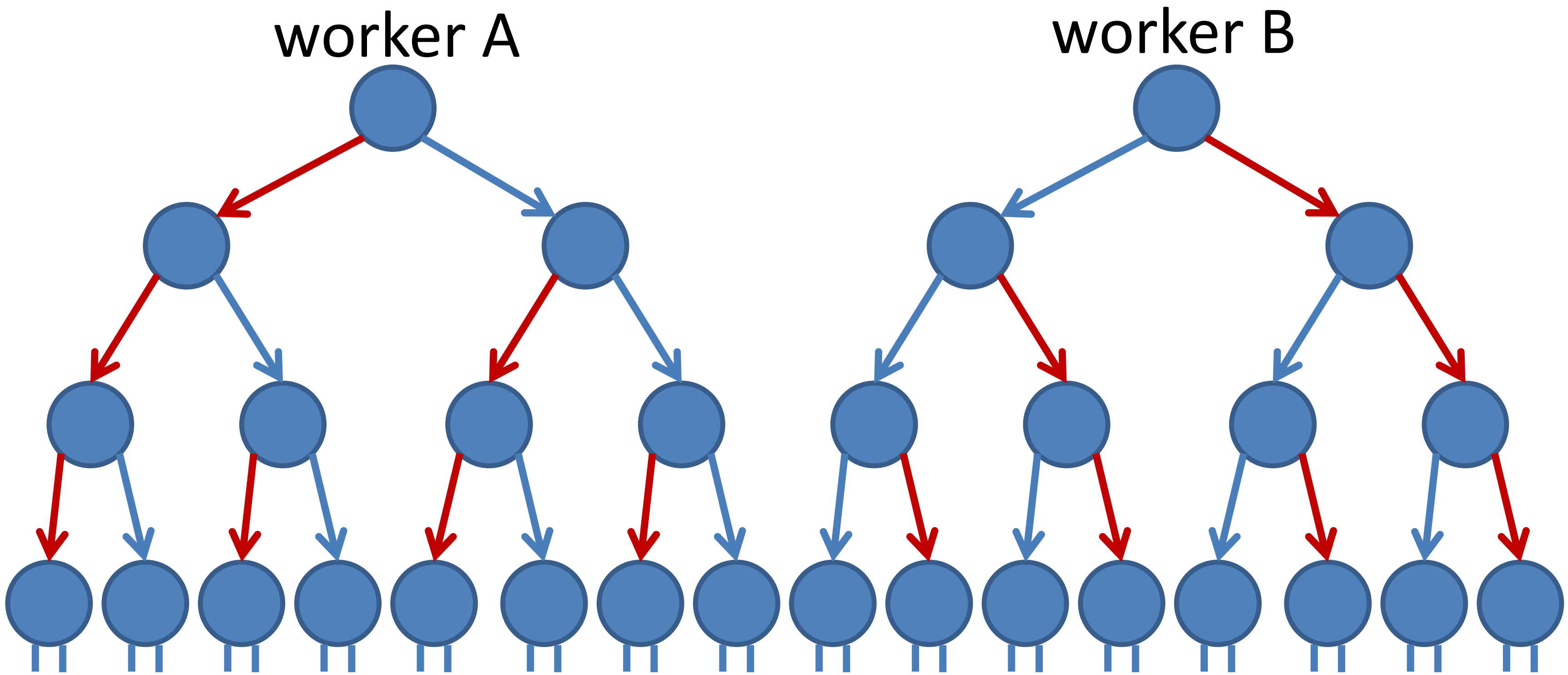
Basic Idea (1/3)

- A worker performs a tree-recursive computation sequentially.



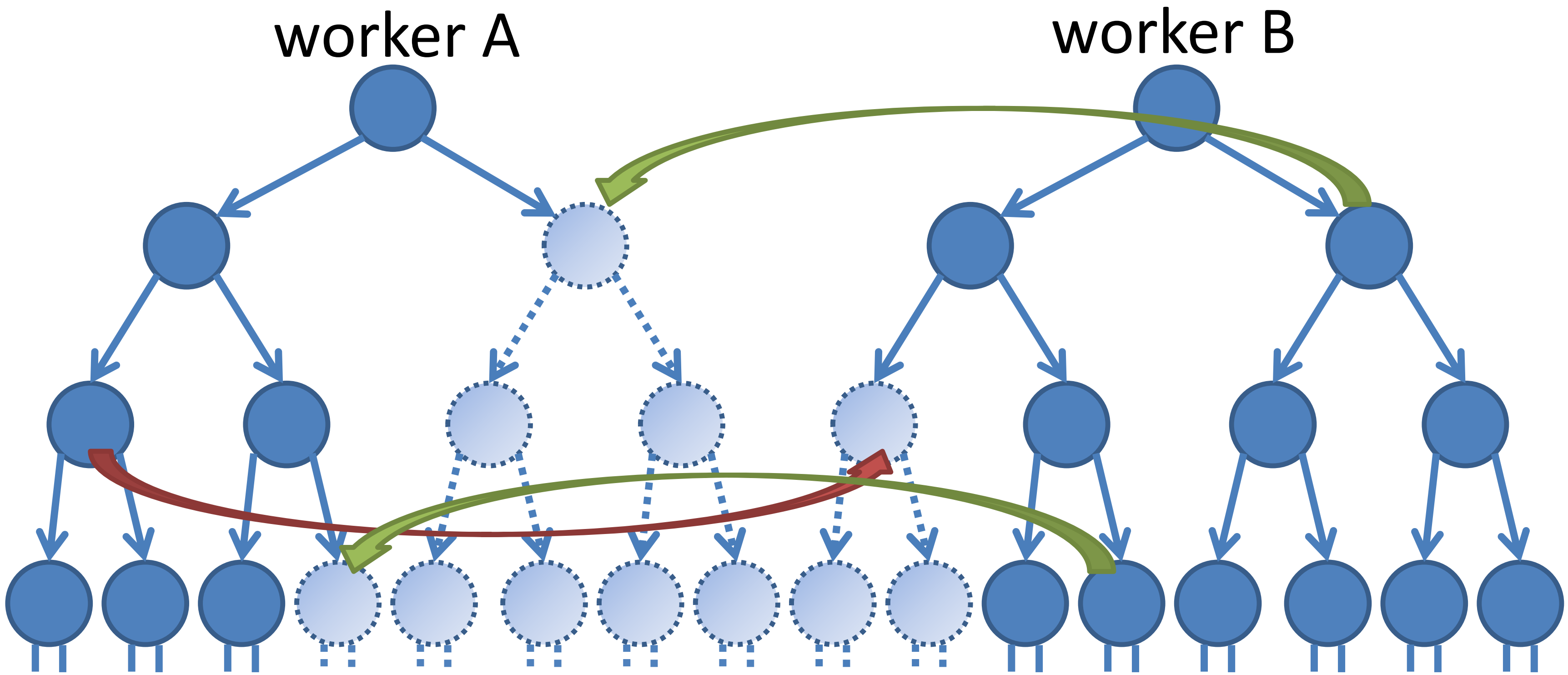
Basic Idea (2/3; redundancy)

- Every worker performs a program with its own planned execution order (SPMO).



Basic Idea (3/3; parallel speedups)

- Exchange partial results for hierarchical omission



Our Approach

- Hierarchical computation
  - Variable-length addresses to identify subcomputations
- The HOPE Language: Directives:
  - Order ($\neq$ spawning concurrent tasks/threads)
  - Access to partial results
- Compiler (translator)
  - into C with nested functions (L-closures [Yasugi+06])
  - We solve dilemmas: 3 execution modes and dynamic switching
- Runtime: Message Mediation Systems (MMSs)
  - Distributed and federated
  - As a local storage and an automatic exchanger of messages of partial results
  - Variable-length addresses to/from which partial results are written/read

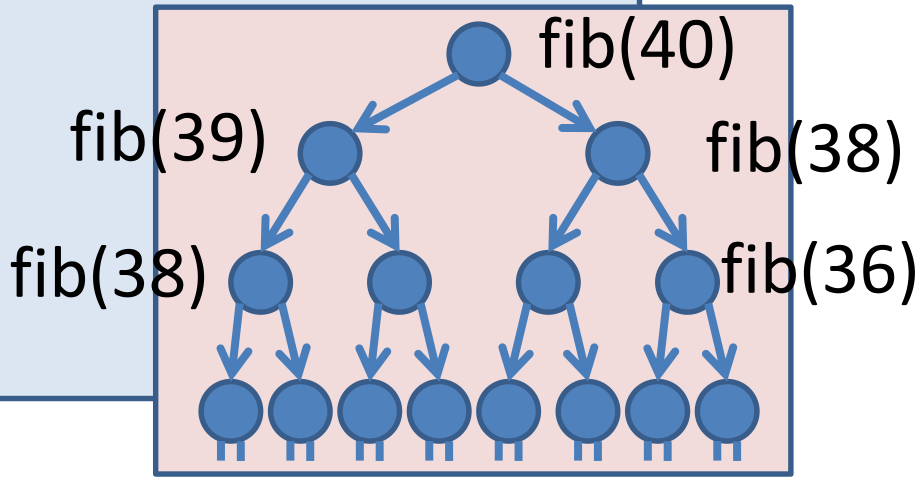
Outline

- Motivating Example
- Our Approach
- The HOPE language
- Implementation
  - The Compiler
  - The Message Mediation System
- Evaluation
- Discussion

Motivating Example

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

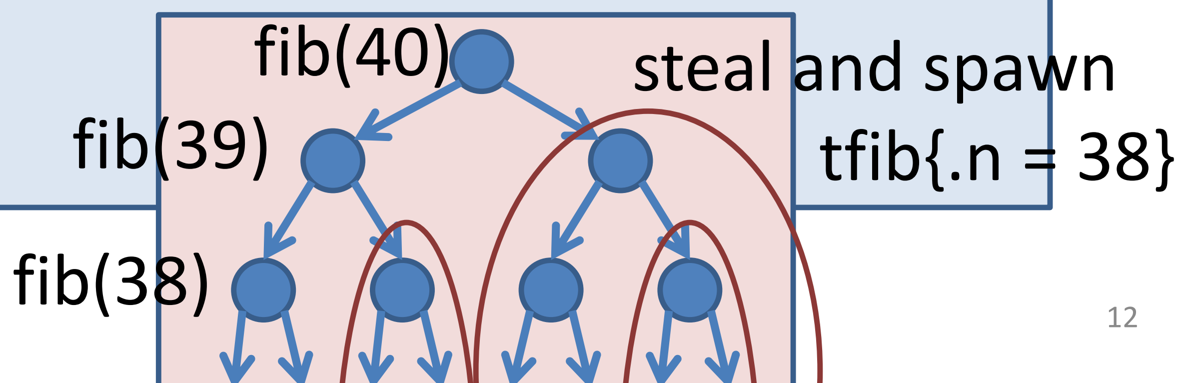
```
int fib(int n) {
    if (n <= 2) return 1;
    int r0, r1;
    r0 = fib(n - 1);
    r1 = fib(n - 2);
    return r0 + r1;
}
```



An existing work-stealing framework: Tascell [Hiraishi,Yasugi+ PPoPP ’09]

- (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;
}
```



Conventional Fault tolerance

- Checkpointing (for work-stealing frameworks)
  - local synchronization between thief and victim
    - Victim can save the stolen task, but
    - Around the root, large part of computation may be lost.
    - The root is a single point of failure.
  - use of non-volatile storage
    - Without high-level semantics, the restarted thief cannot reuse the saved state in the global context.
- Redundant parallel execution
  - no speedups

OUR APPROACH

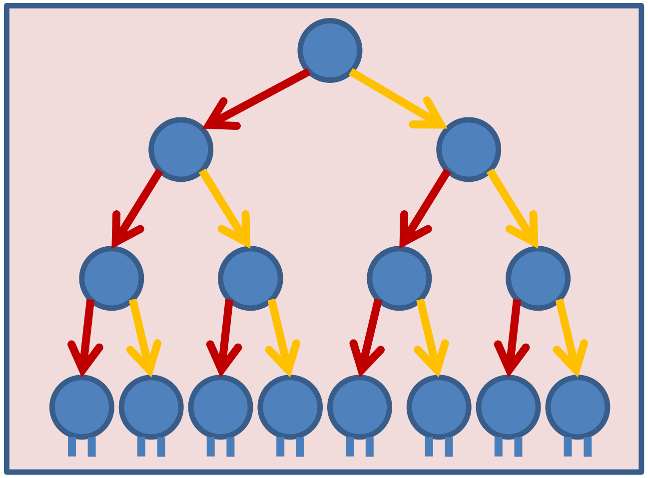
Our Approach

- The HOPE framework
- We propose a new ``hierarchical omission''-based parallel execution model called 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).
  - Workers (and runtime systems) automatically exchange partial results to omit hierarchical subcomputations.

SPMO (Single Program Multiple Order) (each worker's own planned order)

- Hierarchical "multiple-order" computation

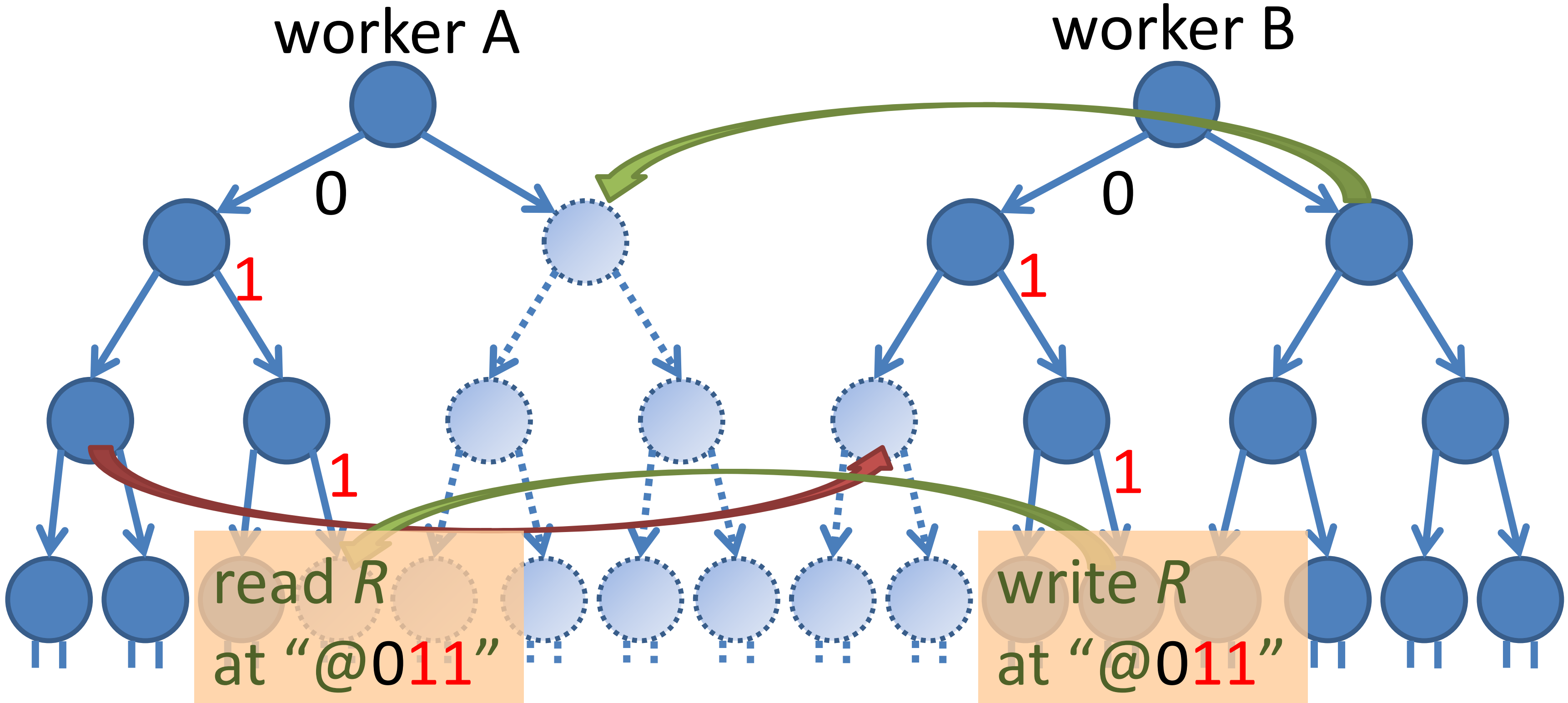
```
int fib (int n) {
    if (n <= 2) return 1;
    int r0, r1;
    Plan and execute
        r0 = fib (n-1);
        r1 = fib (n-2);
    OR
        r1 = fib (n-2);
        r0 = fib (n-1);
    return r0 + r1;
}
```



☺ Cache friendly (for stack)
☹ Order selection cost
☹ Branch-predictor unfriendly

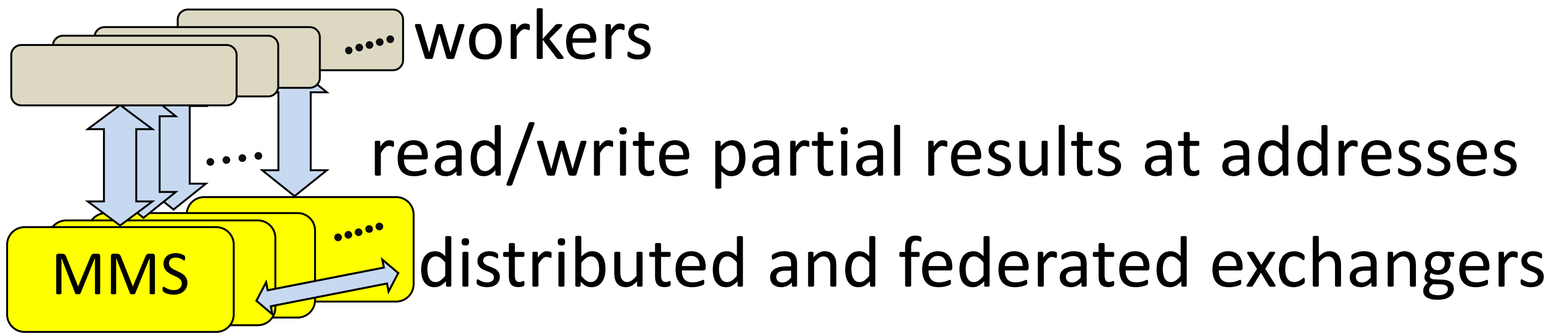
Fault tolerance AND Parallel speedups

- Exchange partial results (as messages) for hierarchical omission



The Message Mediation System (MMS)

- Workers can check/write/read messages. (A worker can read messages in its local MMS if available.)



- Unlike MPI, MMS provides storage functionality.
- Unlike key-value stores, MMS employs variable-length addresses of subcomputations.
- Each message contains the variable-length address and the result of a subcomputation.

THE HOPE LANGUAGE

The HOPE Language

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

- 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);
    #pragma hope omissible result(r1)
    r1 = fib(n - 2);
    return r0 + r1;
}
```

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

IMPLEMENTATION

MMS APIs

- Move cursors over variable-length addresses
- Check if the partial result of a subcomputation is locally available at a cursor
  - Also for *sub*-subcomputations
  - Also for *super*-subcomputations
  - Also if the worker is sufficiently "alone" with respect to other workers (other workers' pre-shared plans)
- Read the available partial result at a cursor
- Write a partial result at a cursor

The HOPE Compiler

- Naive translation with MMS API would involve the following issues to be addressed:
- Dilemma 1 (collaboration cost):
  - ☺ Hierarchical omission of computations
  - ☹ Overheads of writing/checking partial results
  - ☹ Costs of maintaining the “current” address
- Dilemma 2 (multiple-order cost):
  - ☺ SPMO reduces the possibility of overlapping.
  - ☹ Order selection cost
  - ☹ Branch-predictor unfriendly

Solve Dilemma 1 (collaboration cost)

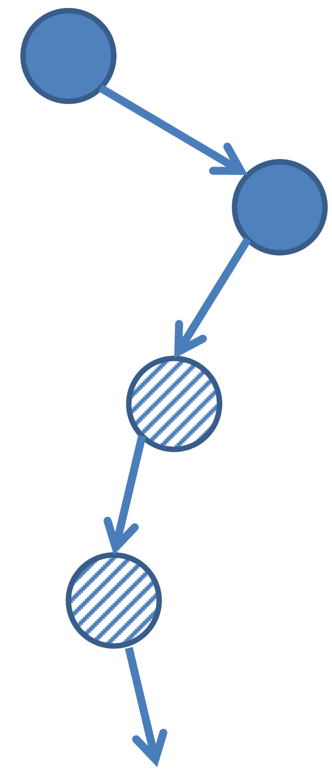
- By using two execution modes

The “check” mode

- writing/checking partial results
- maintaining the “current” variable-length address

The “non-check” mode

- Only SPMO execution
- If the worker is sufficiently “alone”



Solve Dilemma 2 (multiple-order cost)

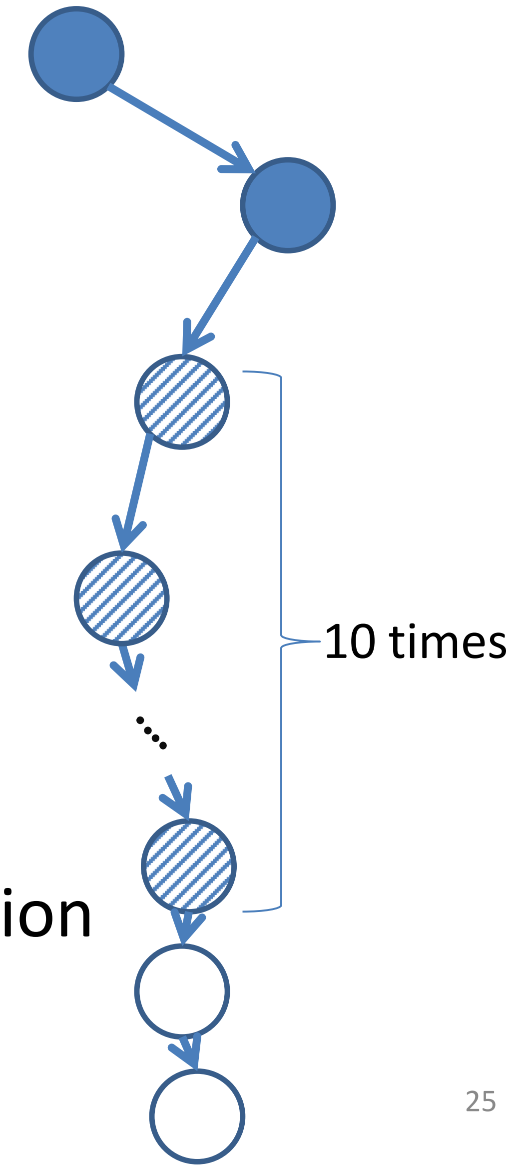
- By using 3 execution modes

The “check” mode

The “non-check” mode

The “work-first” mode

  – Ignore planned order
  – Plain sequential execution
  – After following order selection ten times in hierarchical computation



Dynamic switching with Nested functions (L-closures [Yasugi+06])

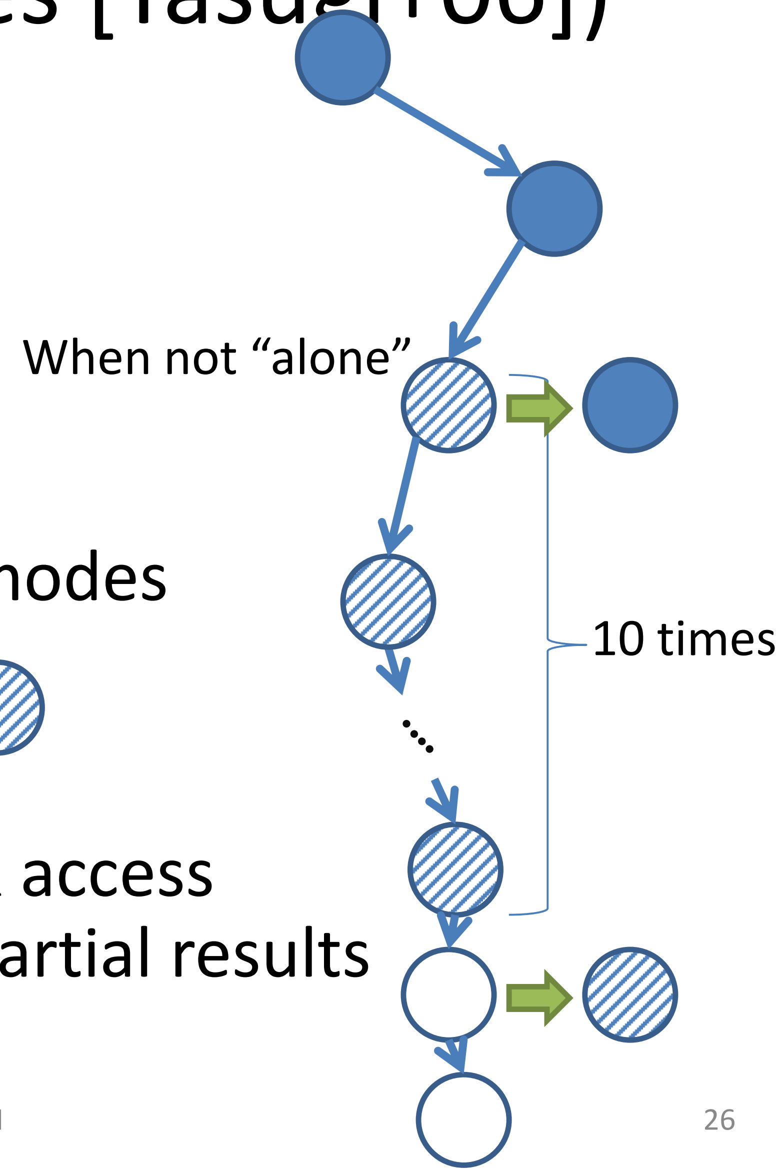
The "check" mode

The "non-check" mode

The "work-first" mode

- Dynamic switching
  - Changes a past choice of modes

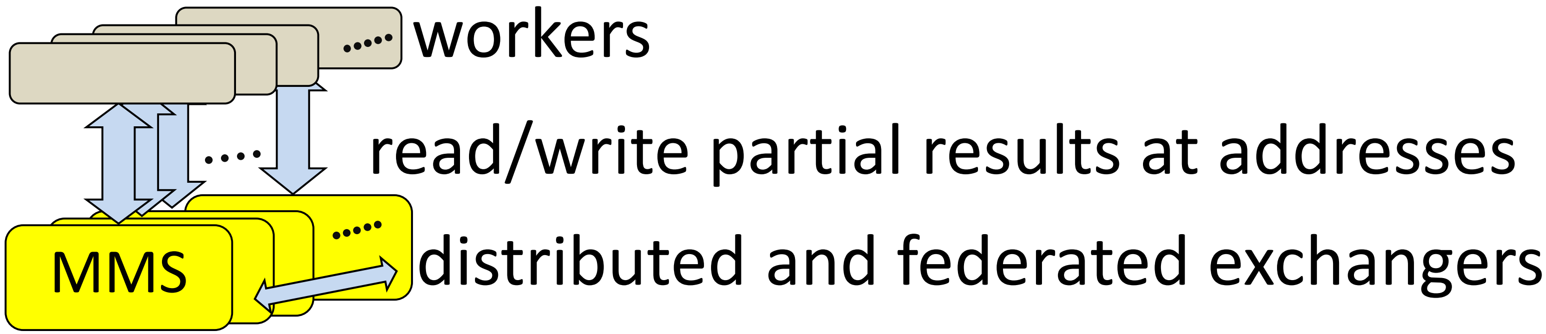
  - Legitimate execution stack access to handle addresses and partial results



Implementation of the Message Mediation System

- Parallel execution and fault tolerance
  - Each SPMO worker process consists of
    - an actual worker (thread)
    - a message mediation subsystem
- Message mediation
  - Subsystems propagate messages only to “relevant” worker processes by estimating participants of a subcomputation based on pre-shared plans
- In each process, worker/receiver threads share a trie for variable-length addresses with appropriate mutex.

Implementation of MMS (cont.)

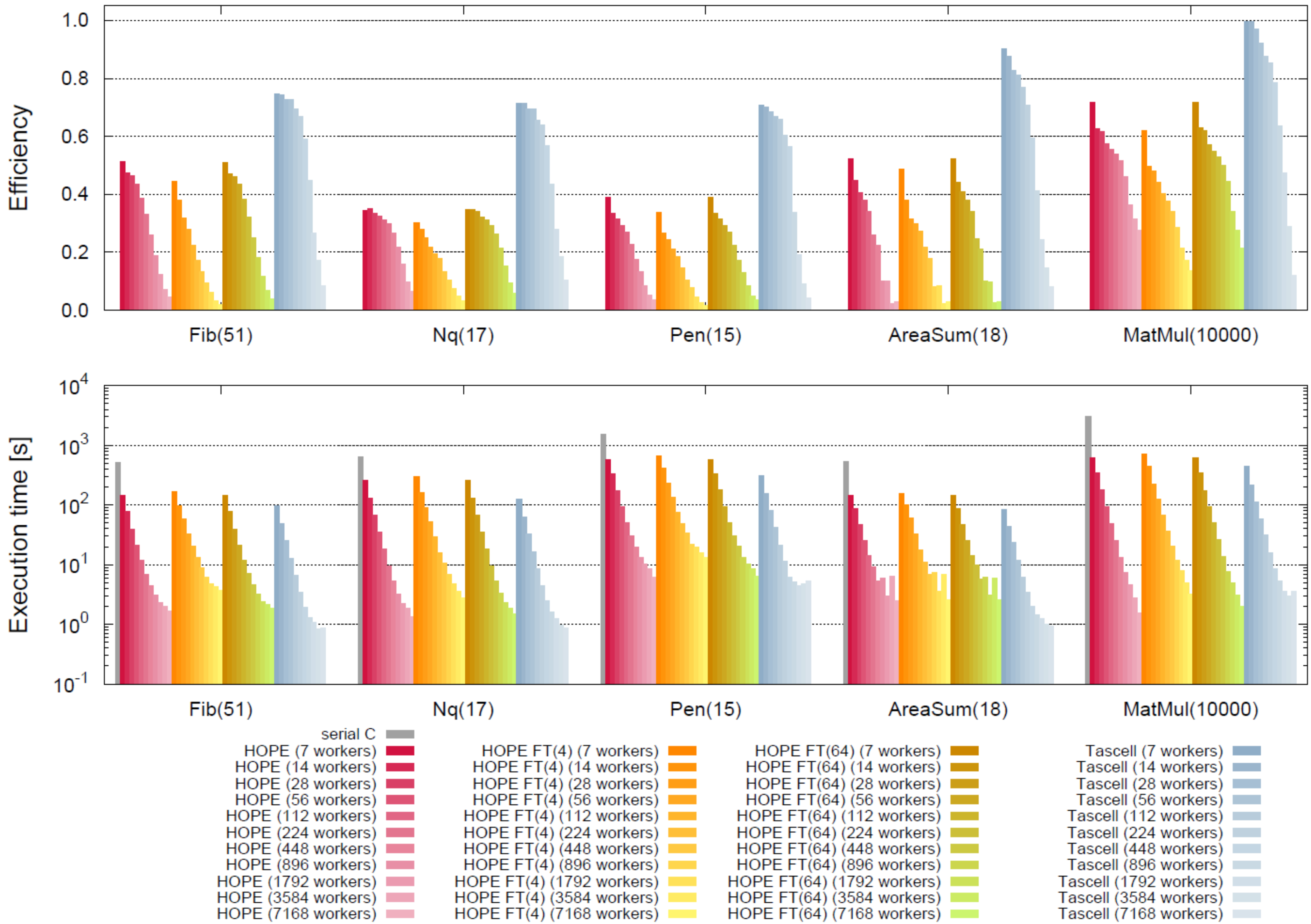


- POSIX threads for concurrency
- MPI for (asynchronous) communication
- Required properties
  - Deadlock avoidance
  - Fairness
  - Fault tolerance

EVALUATION

Environment/Benchmark

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


Efficiency
1.0
0.8
0.6
0.4
0.2
0.0
Fib(51)
Nq(17)
Pen(15)
AreaSum(18)
MatMul(10000)
Execution time [s]
10^4
10^3
10^2
10^1
10^0
10^{-1}
Fib(51)
Nq(17)
Pen(15)
AreaSum(18)
MatMul(10000)
serial C
HOPE (7 workers)
HOPE (14 workers)
HOPE (28 workers)
HOPE (56 workers)
HOPE (112 workers)
HOPE (224 workers)
HOPE (448 workers)
HOPE (896 workers)
HOPE (1792 workers)
HOPE (3584 workers)
HOPE (7168 workers)
HOPE FT(4) (7 workers)
HOPE FT(4) (14 workers)
HOPE FT(4) (28 workers)
HOPE FT(4) (56 workers)
HOPE FT(4) (112 workers)
HOPE FT(4) (224 workers)
HOPE FT(4) (448 workers)
HOPE FT(4) (896 workers)
HOPE FT(4) (1792 workers)
HOPE FT(4) (3584 workers)
HOPE FT(4) (7168 workers)
HOPE FT(64) (7 workers)
HOPE FT(64) (14 workers)
HOPE FT(64) (28 workers)
HOPE FT(64) (56 workers)
HOPE FT(64) (112 workers)
HOPE FT(64) (224 workers)
HOPE FT(64) (448 workers)
HOPE FT(64) (896 workers)
HOPE FT(64) (1792 workers)
HOPE FT(64) (3584 workers)
HOPE FT(64) (7168 workers)
Tascell (7 workers)
Tascell (14 workers)
Tascell (28 workers)
Tascell (56 workers)
Tascell (112 workers)
Tascell (224 workers)
Tascell (448 workers)
Tascell (896 workers)
Tascell (1792 workers)
Tascell (3584 workers)
Tascell (7168 workers)

DISCUSSION

A Limitation

- ☺ Redundantly assigning a value to a location
  - "assigning a value to a location" is idempotent.
- ☹ Redundantly applying an effect to a location
  - Accumulators:

```
x = f(n);
a[x.i] += x.v;
```

  - NOTE: |=, &=, max= is OK
  - NOTE: worker-local effect is OK
    (cf. workspaces for *n*-queens)

Related work

- Checkpointing methods
  - save all states.
    - Complex algorithm for distributed snapshot
    - Our approach only saves partial results.
  - local synchronization between thief and victim
    - Large part of computation may be lost.
    - The root is a single point of failure.
  - use of non-volatile storage
    - Without high-level semantics, the restarted thief cannot reuse the saved state in the global context.

Related work

- Redundant parallel execution
  - Parallel SAT solvers: Lemma exchange
  - Parallel (game) tree search: transposition tables
  - Memoization
  - SPMD: (e.g., XcalableMP)
    - Communication vs. Computation
- Failure-oblivious computing
  - Our approach is OK even when only one worker remains