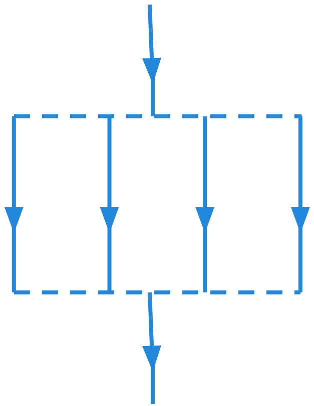
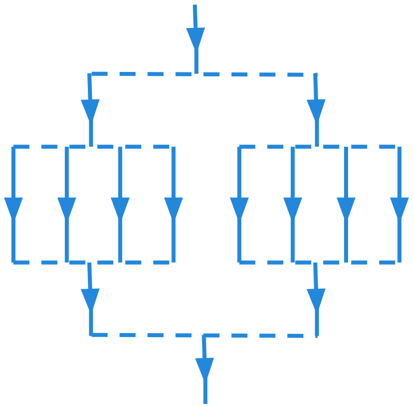
# NestedMP: Taming Complex Configuration Space of Degree of Parallelism for Nested-Parallel Programs

Jiangzhou He, Wenguang Chen, Zhizhong Tang
Tsinghua University

# Nested-Parallel Applications

- Applications with multi-level parallelism





# Why Nested Parallelism for NUMA is necessary

- Necessary for best performance
  - Outer-parallel only: hard to utilize all cores, poor load balance
  - Inner-parallel only: too fine grain, too much context-switch overhead
- Applications benefits from nested-parallelism
  - Computational Fluid Dynamics Applications
  - Derivation Computation Application
  - Strassen Matrix-Multiplication Algorithm
  - Cooley-Tukey Fast Fourier Transformation Algorithm
  - Multisort Algorithm

# Challenge for Nested-Parallel Programming: Configuration for Degree of Parallelism (1)

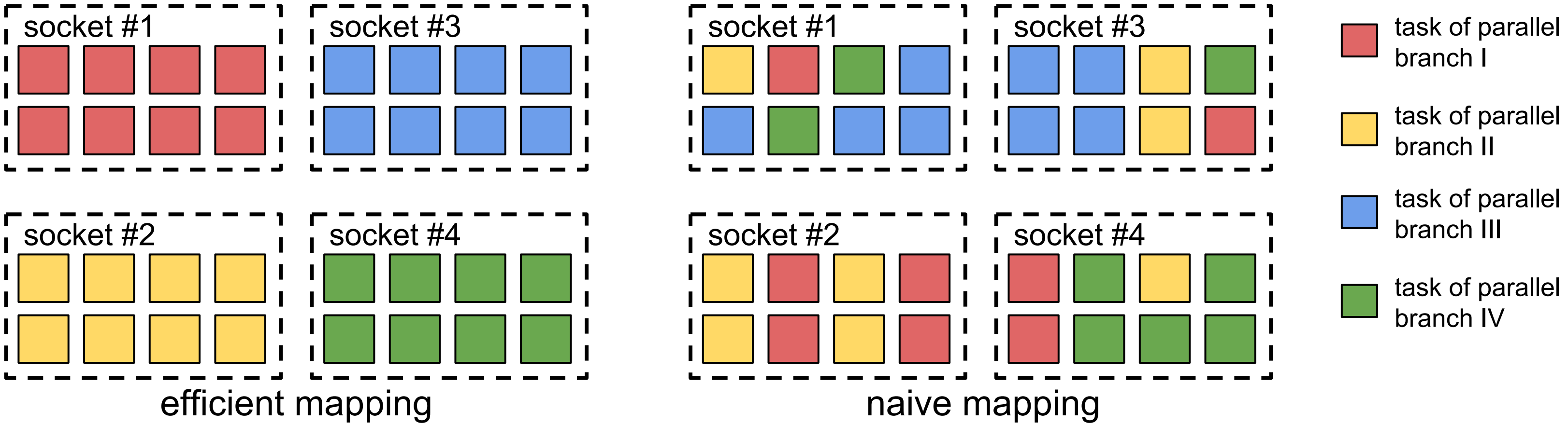
- Configuration space is complex
  - Should outer-level or inner-level have more degree of parallelism?
    - If we have 8 cores, possible configuration may be 1x8, 2x4, 4x2, 8x1
  - Second-level parallelism may be asymmetrical
    - When the first-level parallelism is fixed to 2,<br>1+7, 2+6, ……, 7+1 are possible
  - Different phases of an application may need different configuration

# Challenge for Nested-Parallel Programming: Configuration for Degree of Parallelism (2)

- Configuration should be adaptive
  - Parallel programs should work on processors with different core hierarchy
  - Parallel subroutines may be invoked either exclusively or parallel with other sequential/parallel task

# Challenge for Nested-Parallel Programming: Locality Issue

- Performance varies by different task-core mapping schemas
- Example: NPB-MZ running on 4-way 8-core SandyBridge server, performance varies by 135% for different mapping schemas



# Current Method of Configuring Degree of Parallelism in OpenMP

- Centralized configuration: OMP_NUM_THREADS
  - Poor expressiveness
  - Low-level details is opaque to top-level application programmer / user
- Local configuration
  - Not easy to compute configuration in an adaptive way
  - Runtime lacks global information for optimal task-core mapping
- Fine-grained tasks and queue-based dynamic scheduling
  - Performance loss due to locality issue

# Our Approach

- Underlying problem of local configuration mechanism
  - Degree of parallelism configured by concrete value
  - Everything about degree of parallelism is configured at bottom level
- We designed NestedMP

# Mechanism of NestedMP

- Allocation of Threads
  - Available threads is resource, propagating along the task tree
  - All threads are available threads for root task
  - Once entering a parallel region, available threads of current task are allocated to subtasks
  - Available threads of finished tasks can be reallocated by parent task
- Top-down propagation makes runtime system aware of global information
- Programmers control the **policy** to propagate available threads rather than concrete numbers

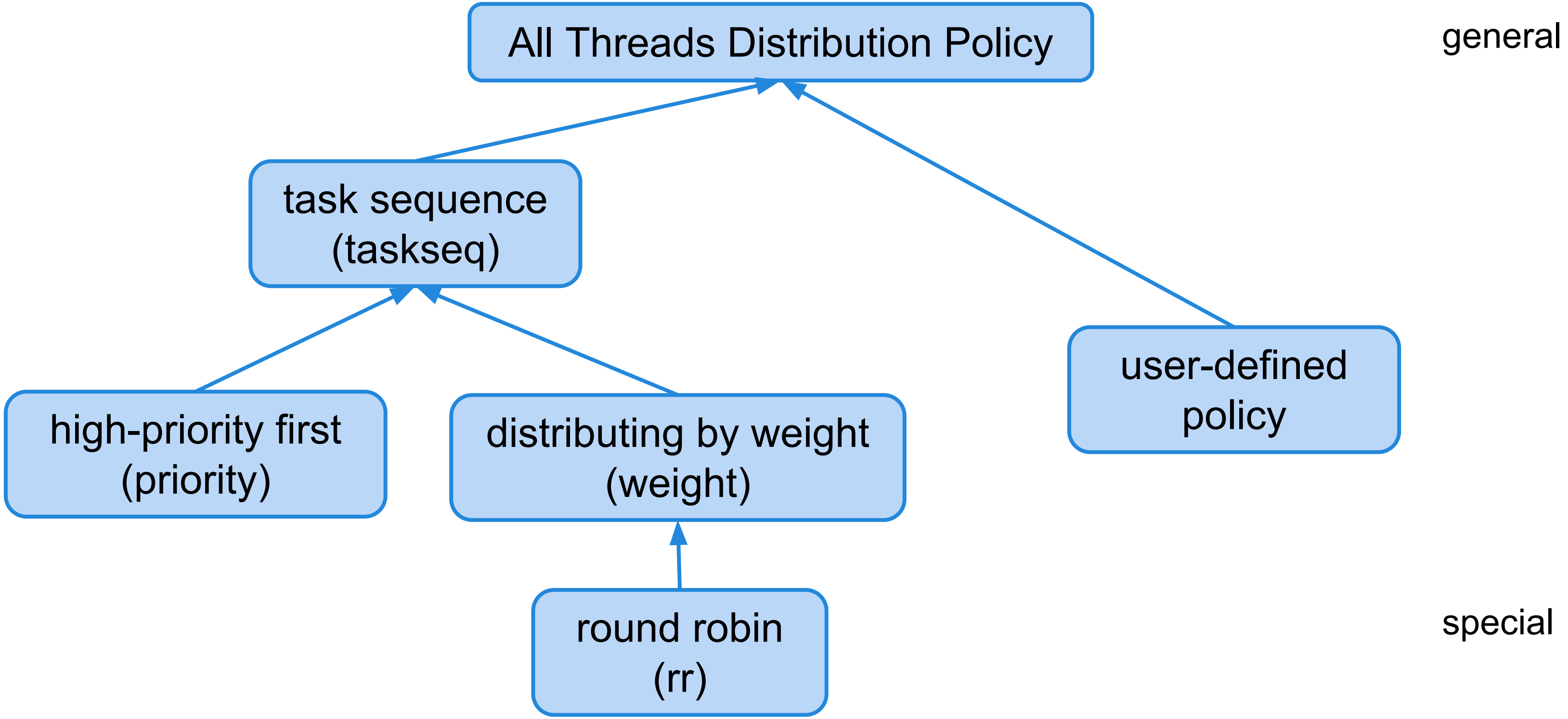
# Policies in NestedMP

- Threads Distribution Policy
  - Determine how to allocate/reallocated available threads among subtasks
- Threads Requirement Policy
  - Subtask decide number of threads which is actually required (rest threads can be reallocated by parent)
  - Task can free available threads by adjust threads requirement policy during execution
- NestedMP has builtin policies, and it also provides interface for users to extend

# Example: Parallel Sort

```
#pragma omp req num_threads(power:2)
void multi_sort(int *s, int n) {
  if (omp_get_num_avail_threads() > 1) {
    int m = n / 2;
#pragma omp parallel sections dist(rr)
#pragma omp req num(seq)
    {
#pragma omp section
      multi_sort(s, m);
#pragma omp section
      multi_sort(s + m, n - m);
    }
    merge(s, m, s + m, n - m);
  } else {
    sequential_sort(s, n);
  }
}
```

# Kinds of Threads Distribution Policy



# Task Sequence (1)

- **Task sequence** is the most general builtin way to expressthreads distribution policy
- A task sequnce is a finite or infinite sequence of tasks

```
t1 t2 t3 t1 t1 t2 t3 t1 t1 t2 t3 t1 ......
```

When 12 threads are available: t1 gets 6, t2 gets 3, t3 gets 3

When 6 threads are available: t1 gets 3, t2 gets 2, t3 gets 1

When 2 threads are available: t1 gets 1, t2 gets 1
(next available thread is for t3, then t1, …)

# Task Sequence (2)

- Task sequence can be expressed by task sequence expression
  - Example: `(t1 t2 t3 t1)*, t1 (t2 t3)*`
- Expressiveness of task sequence
  - high-priority-first is a special case:
    - Example: `(t1 t2)* t3*` (Priority: t1 == t2 > t3)
  - distributing-by-weight is a special case:
    - Example: `(t1 t2 t3 t1)*` (Weight: 2:1:1)
  - other example:
    - `t1 (t2 t3)*` means first thread for t1, rest distributing even to t2 and t3

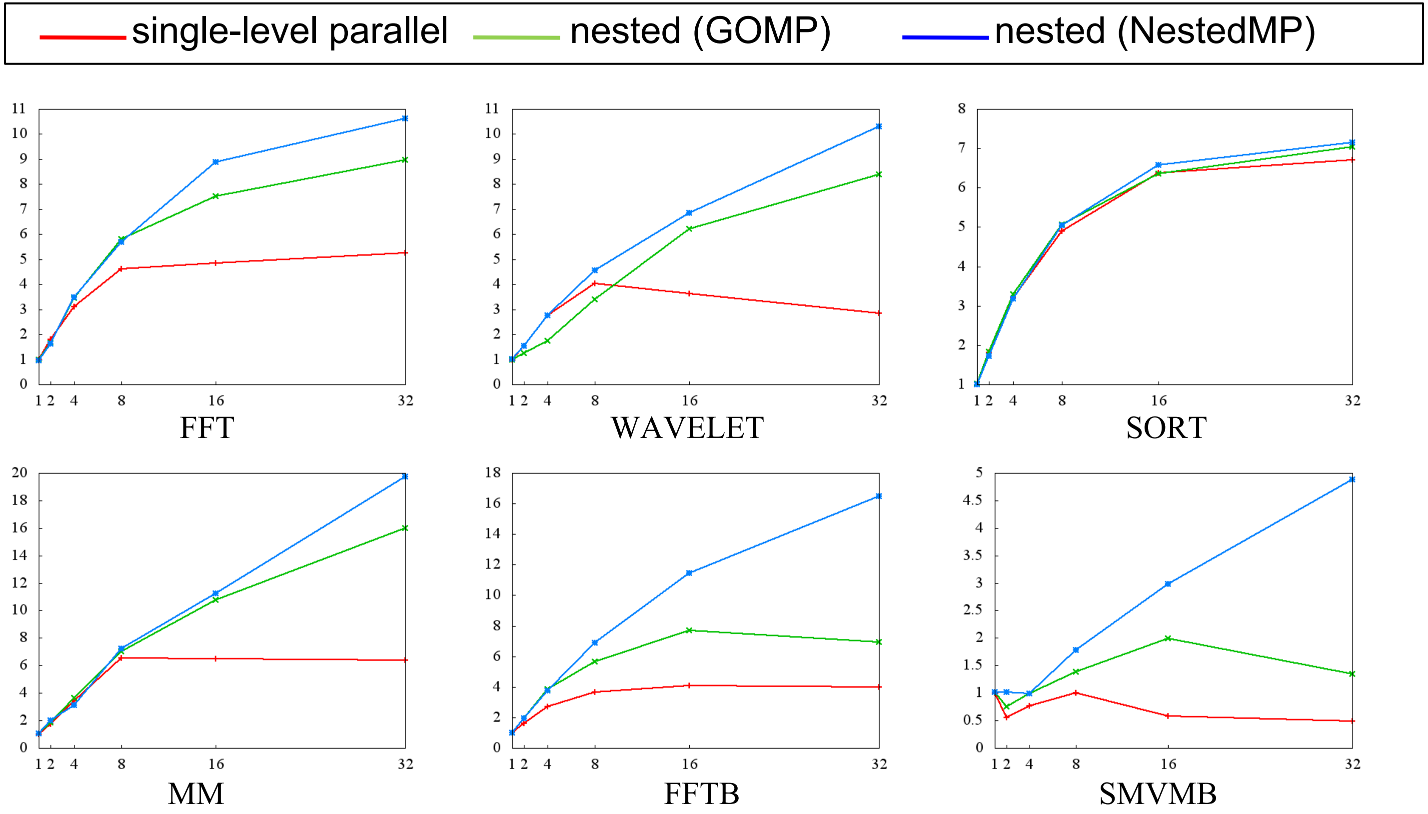
# Threads Requirement Policy

- Requirement for threads number
  - any: accept any available threads
  - seq: accept one and only one thread
  - constant: number of acceptable thread is upper-bounded by a constant
  - power: number of acceptable threads is 1 or $KP^n$ (e.g. multisort accepts $2^n$ threads, here $K$ = 1, $P$ = 2)
- Requirement for locality
  - locality compactness level: host, socket or core
  - locality preference: compact, neutral or spread

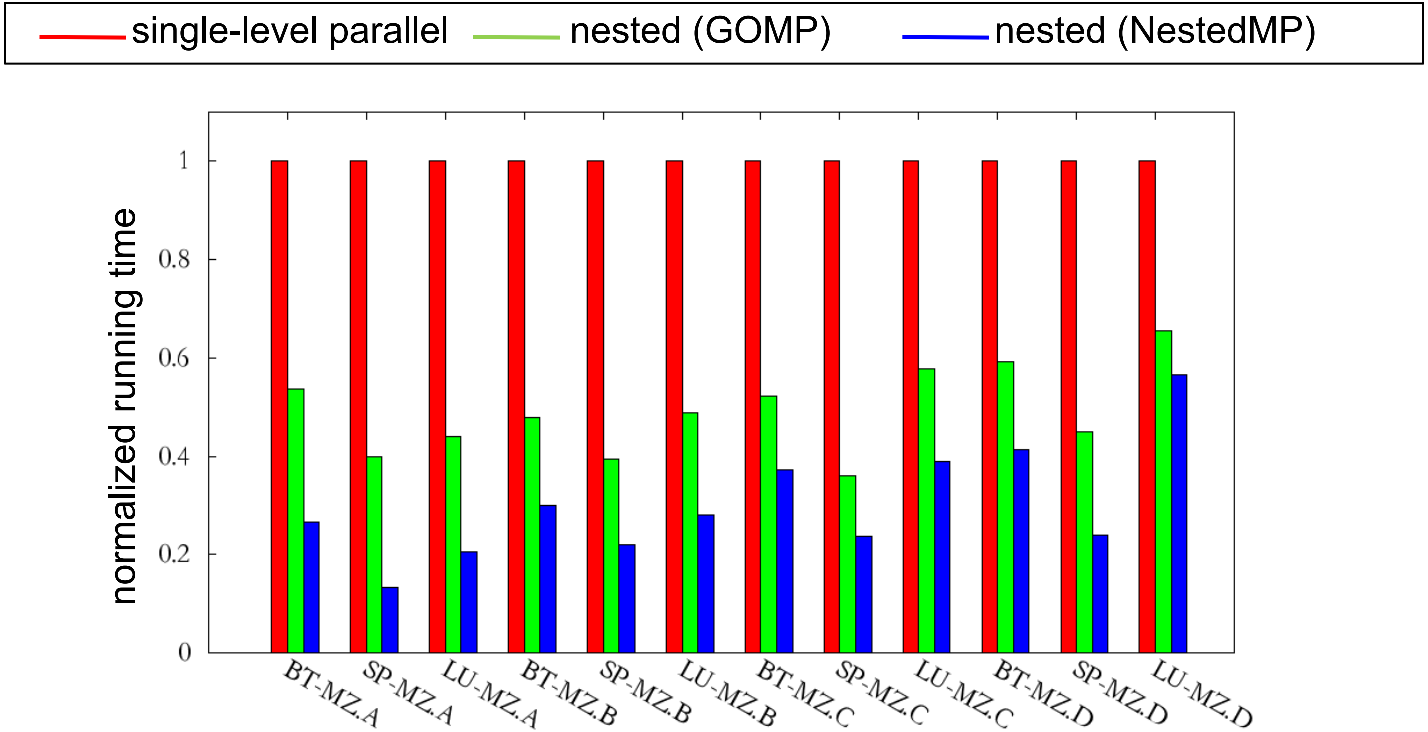
# Evaluation: Benchmarks

- Micro-benchmarks
  - FFT
  - 2D Wavelet Transform
  - Multisort
  - Matrix Multiplication
  - FFT in Batch
  - Sparse Matrix Vector Multiplication in Batch
- NBP-MZ: Scale A, B, C, D

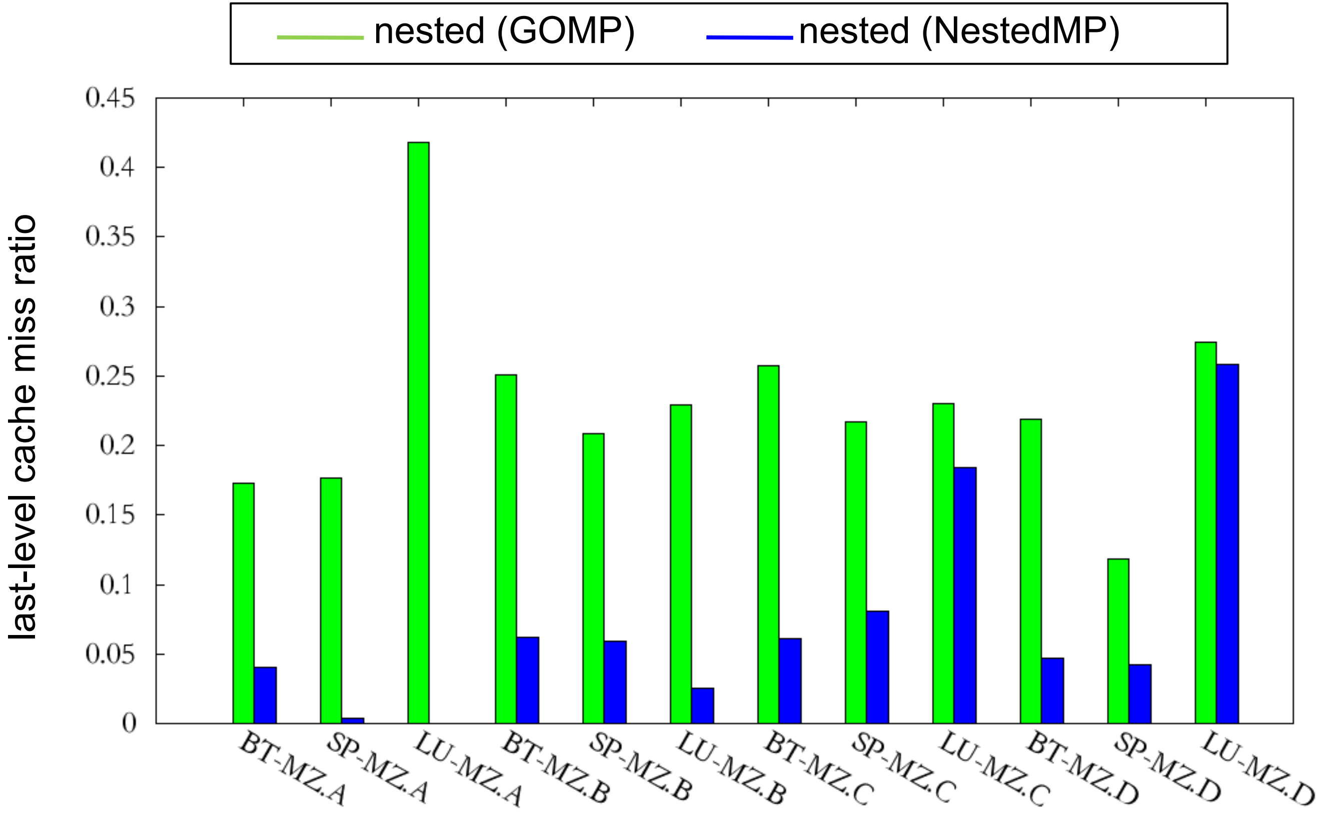
# Speedup of Micro-Benchmarks



# NPB-MZ: Normalized Running Time



# NPB-MZ: Last-level Cache Miss Ratio



# Conclusion

- NestedMP
  - Easier to configure degree of parallelism
  - Configuration is adaptive for different context
  - Expose more information earlier for runtime, so achieved better performance