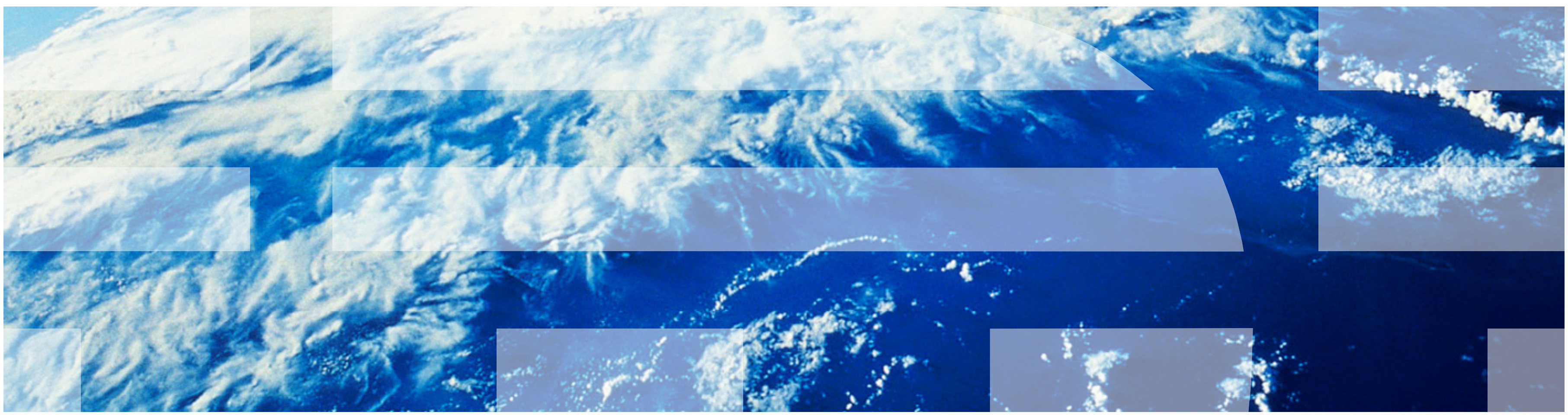
Alexandre Eichenberger - Kevin O' Brien

6/26/12

A Lightweight OpenMP Runtime

-- OpenMP for Exascale Architectures --



T.J. Watson, IBM Research

Goals

- **Thread-rich computing environments are becoming more prevalent**
  - more computing power, more threads
  - less memory relative to compute

- **There is parallelism, it comes in many forms**
  - hybrid MPI - OpenMP parallelism
  - mixed mode OpenMP / Pthread parallelism
  - nested OpenMP parallelism

- **Have to exploit parallelism efficiently**
  - providing ease of use for casual programmers
  - providing full control for power programmers
  - providing timing feedback

Objectives Of Lightweight-OpenMP Runtime

- **Handle more threads**
  - lower OpenMP overheads
    - lower scalar overheads (Amdal's law)
    - better scaling of overheads (more threads)
  - develop new algorithms inside research runtime

- **Handle nested parallelism: more control with thread affinity**
  - more user input on how to map computation to threads
    - currently: no affinity support provided by user
  - proposed a new thread-affinity to OpenMP standard committee
  - contributed reference implementation in research runtime

- **Todo: Provide timing feedback**
  - user want to know where is the time spent
  - feedback at little overheads

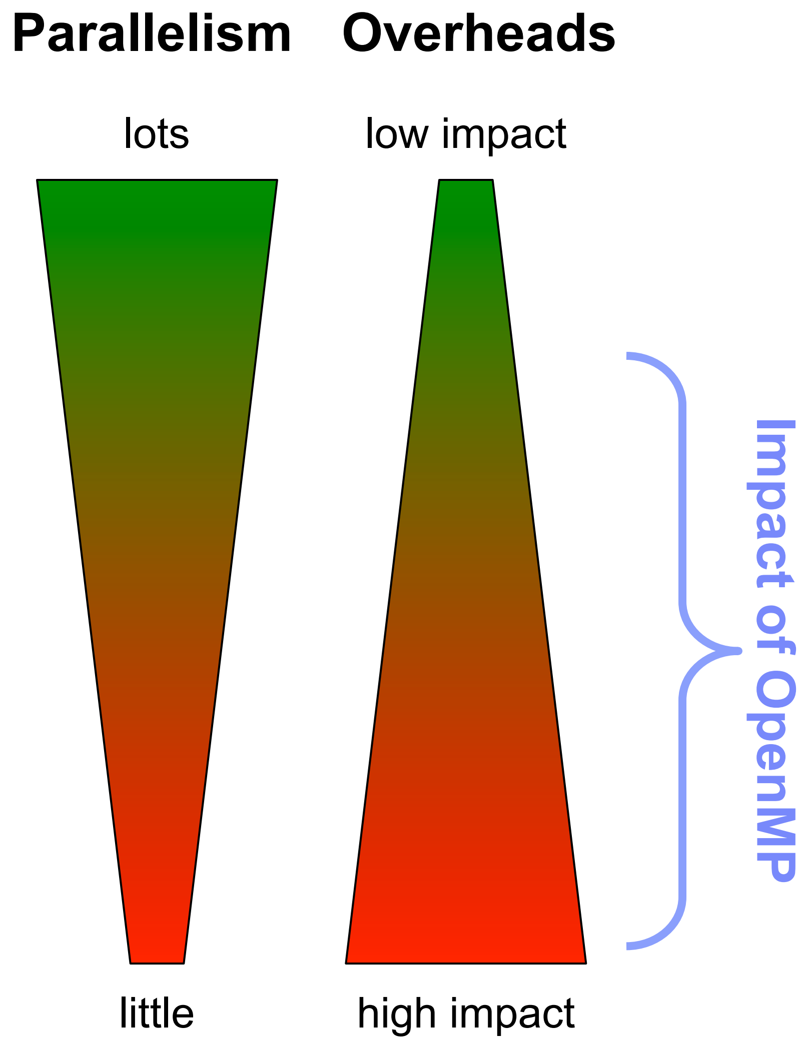
Part 1: Handle more threads

- **Impact of overheads**
- **Approach for near constant-time parallel-region creation**
- **Results on BGQ**

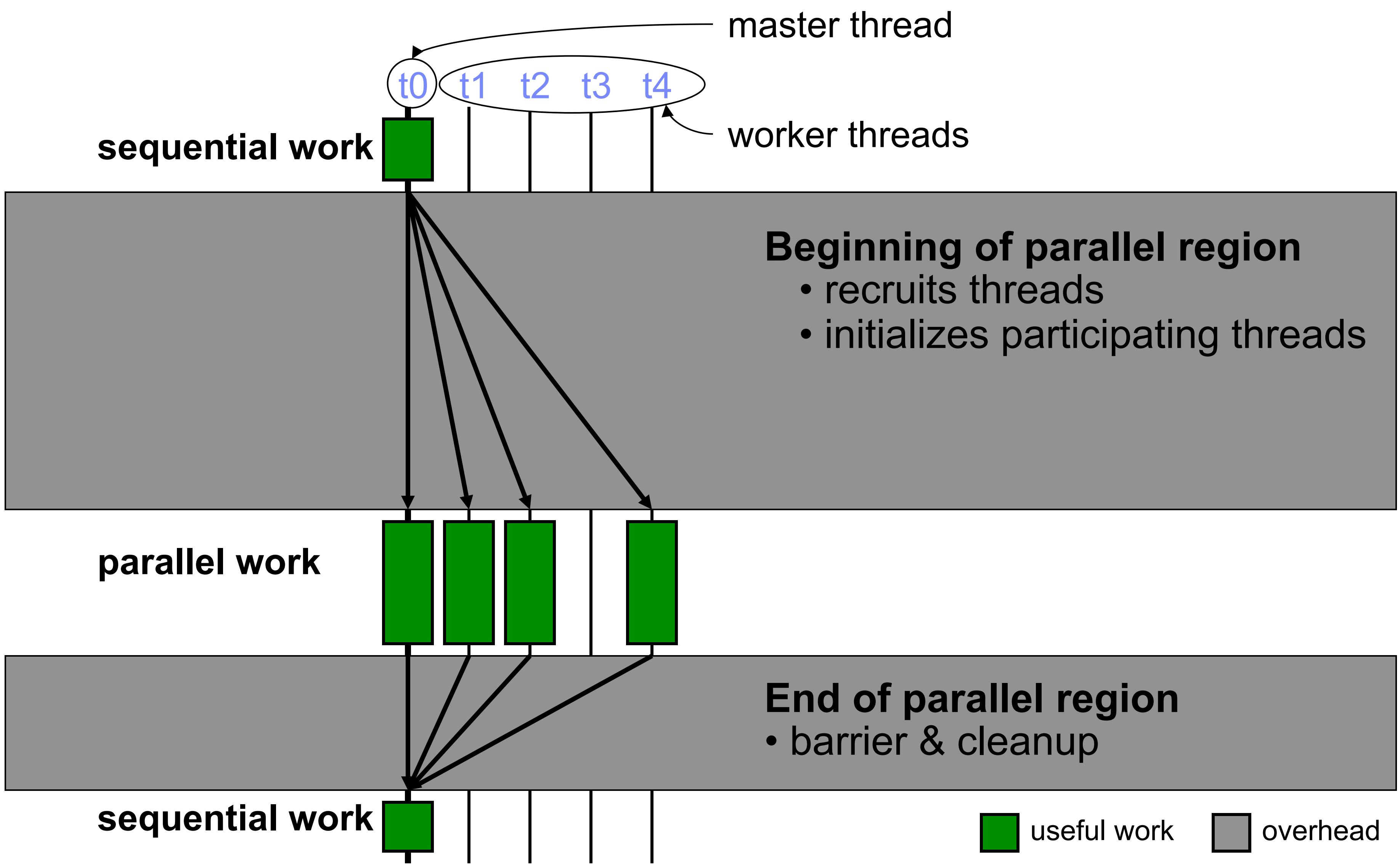
Impact of Overhead in Prevalent Threading Model

Programming Model

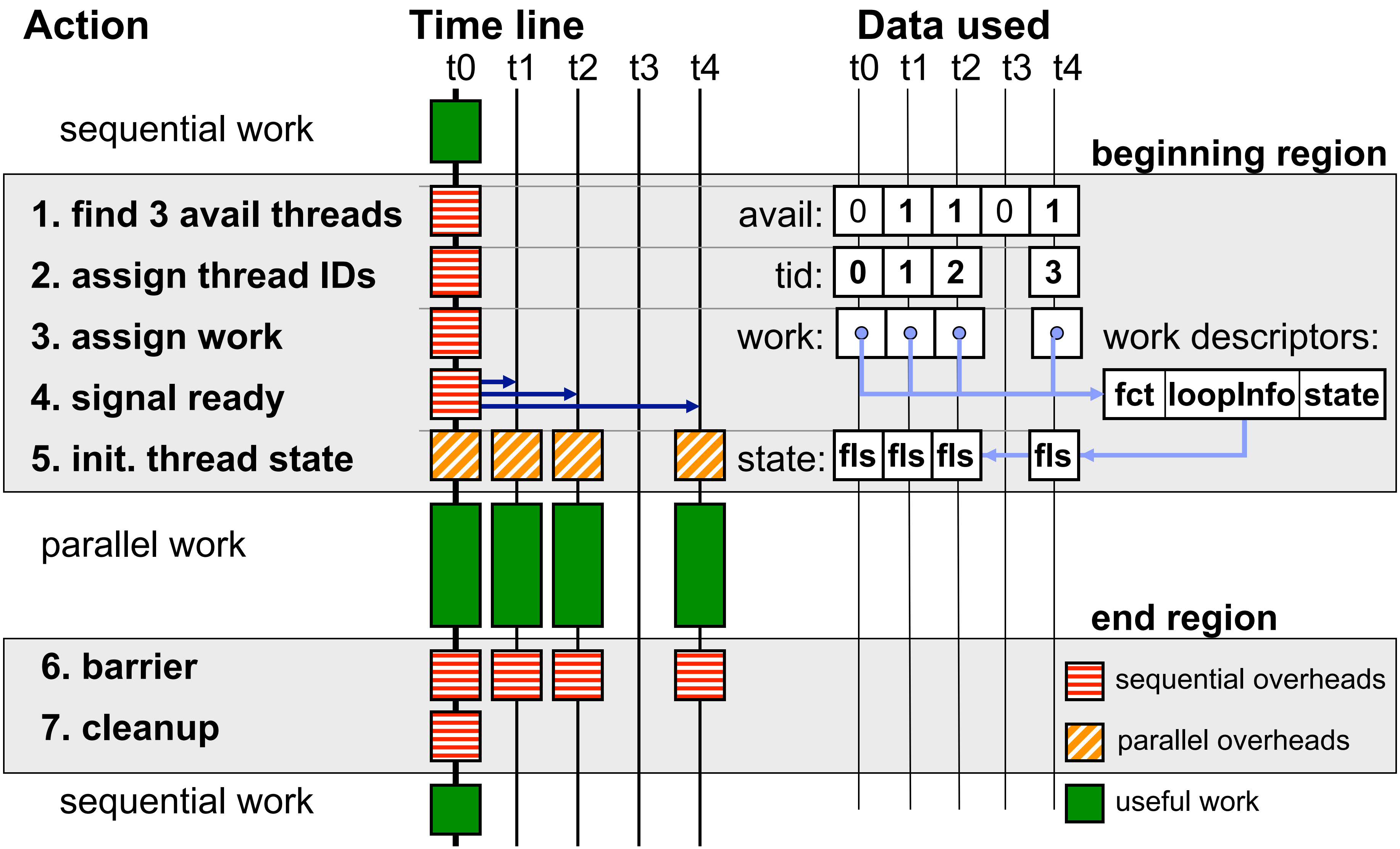
- **MPI**
  - distributed process across/within nodes
  - explicit user-managed communication
- **Coarse-grain Parallel (OpenMP/Auto)**
  - shared memory within nodes/cores
  - for outer parallel-loops
- **Fine-grain Parallel (OpenMP/Auto)**
  - shared memory within cores/nodes
  - for inner parallel-loops



Basic OpenMP Operation: Parallel Region



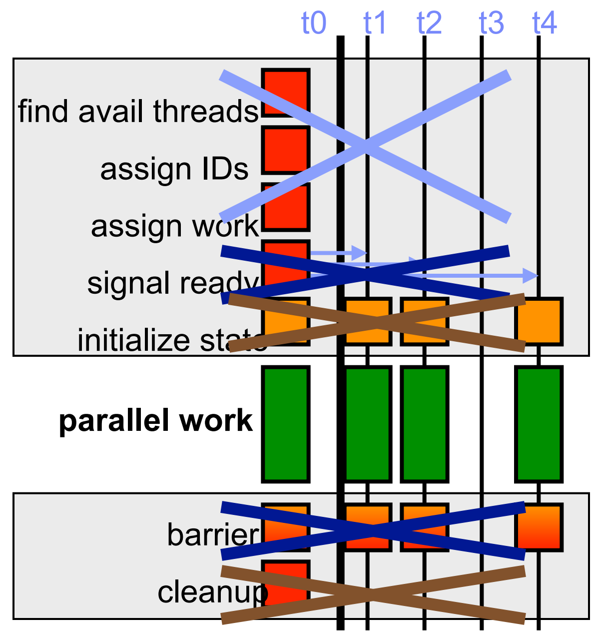
Source of Overheads, Due to OpenMP Standard



Optimized OpenMP Runtime Design

Systematic re-design to lower overheads

- **Eliminate sequential overhead**
  - reuse previous thread allocations
  - in practice, near 100% hit
- **Extremely compact state**
  - minimize initialization/cleanup
- **Use hardware support**
  - atomic instructions (atomic increment / xor)



Optimization Guiding Principles

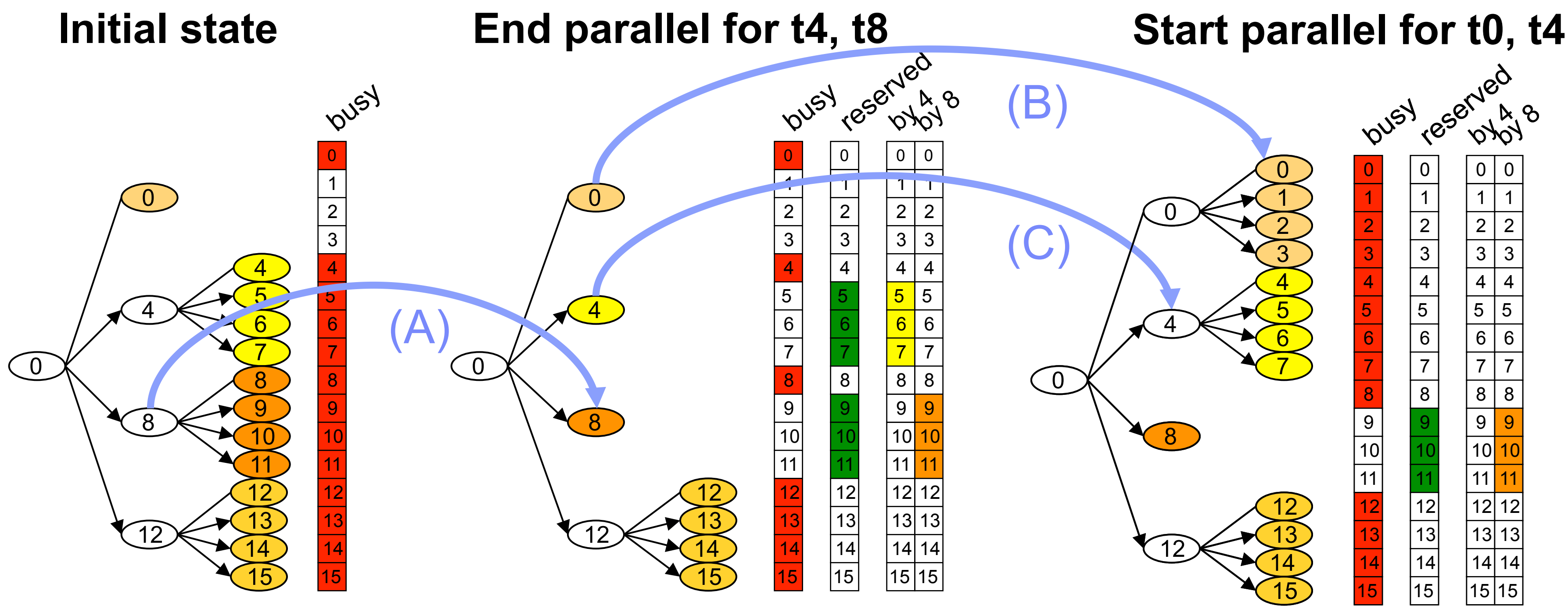
- **Cache configurations to eliminate computation & communication**
  – reuse as much as possible when nothing has changed

- **Minimum locking**
  – one lock for protecting thread allocation data structure
  – locked only on thread recruiting / freeing
  – rest use atomic operations

- **Use global state sparingly**
  – work descriptor is only used for parallel region
  – most other OpenMP constructs use no work descriptors

- **Allocate state statically, initialize mostly statically**
  – barriers use counters initialized when initializing OpenMP
  – some local state is only initialized on first use

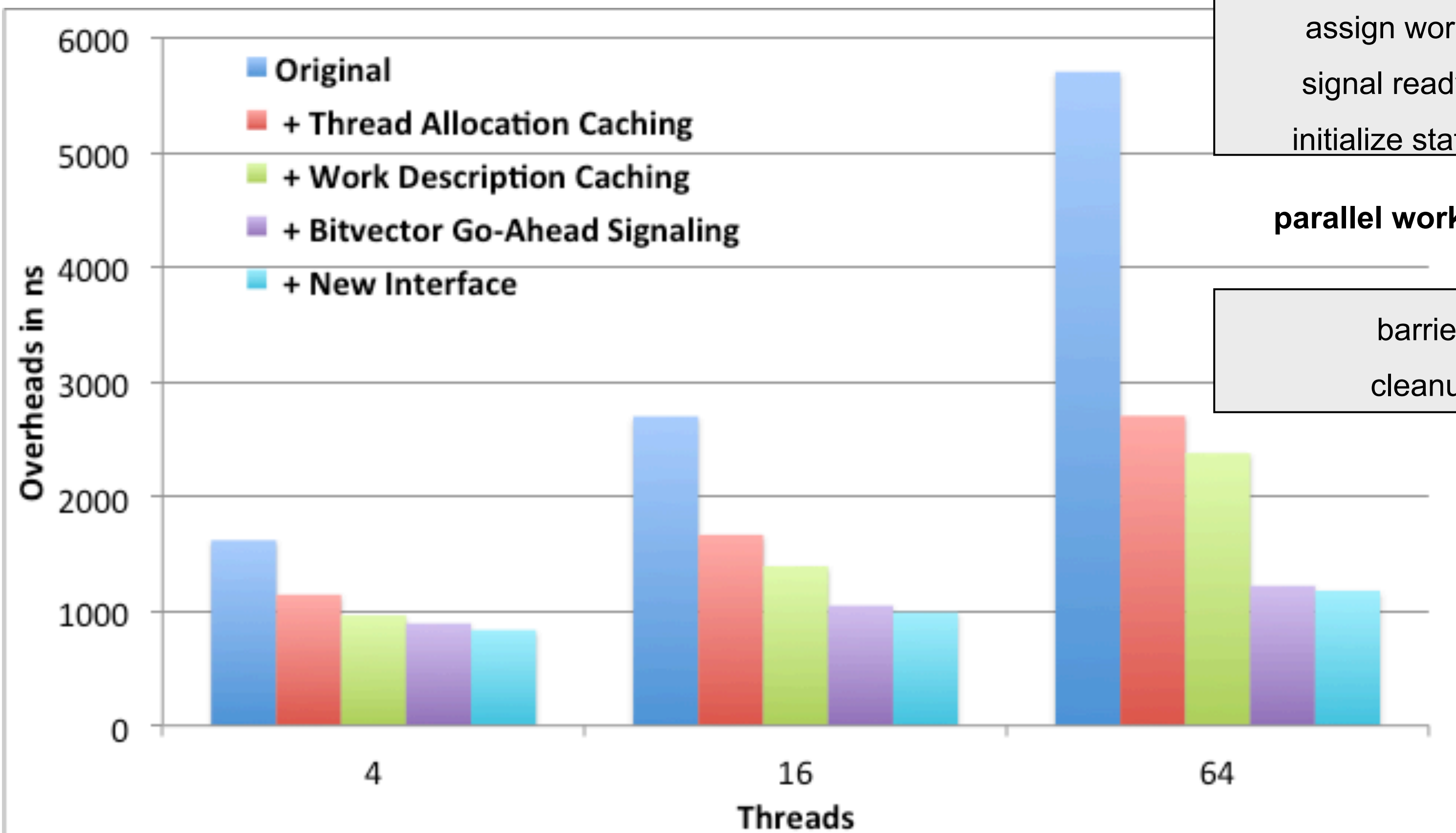
Example: Caching Worker Configurations

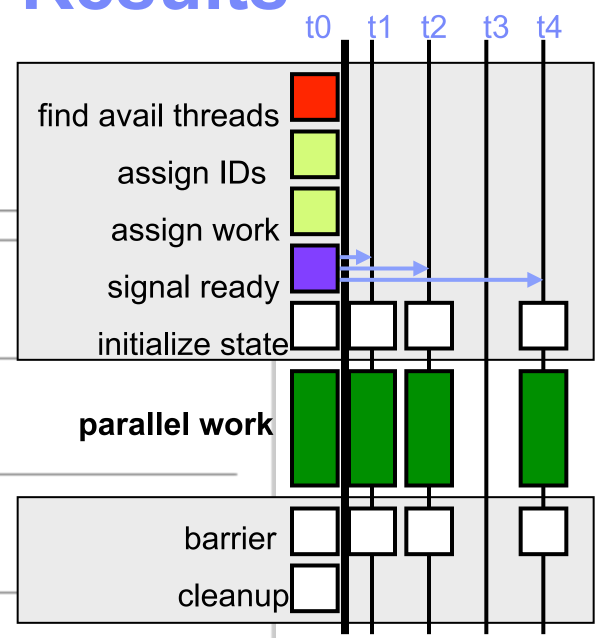


- **When freeing workers**
  - leave workers in reserved state (A: end t8)
- **When recruiting workers**
  - avoid stealing workers that were reserved by others (B: start t0)
  - aim at reusing workers that were previously reserved by this master (C: start t4)

OpenMP Micro-Benchmark (EPCC) Results

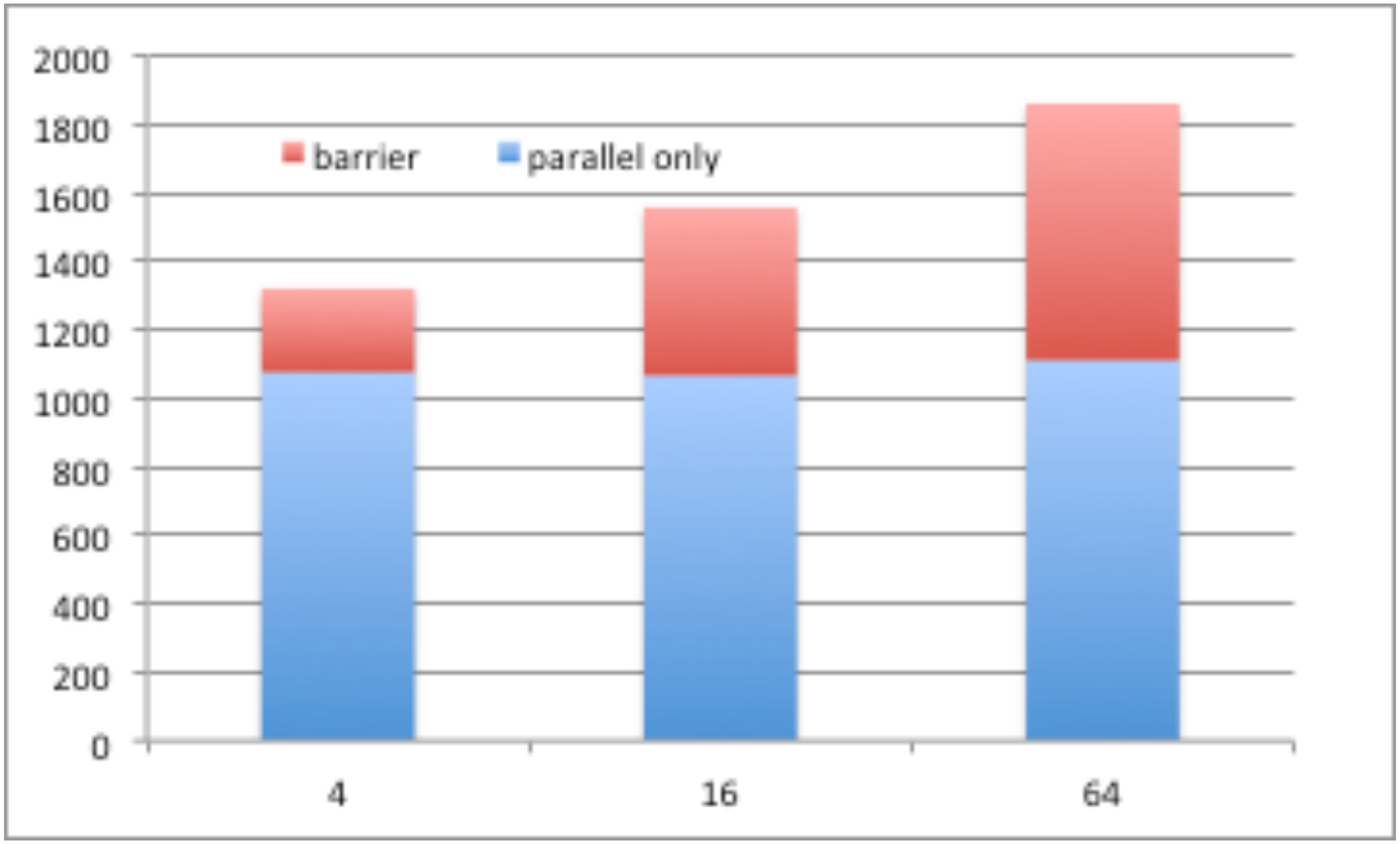
- **Contributions of individual optimizations**





Overhead Scaling for Parallel Region (ECPP)

- **Nearly constant overhead over wide range of thread counts**



LOMP is an experimental runtime that implements a subset of all OpenMP functionality. Performance will be impacted until full functionality is provided

Observations

- **Creating a parallel region with 4 to 64 threads**
  - overhead are now reduced to below 2K cycles
  - preliminary numbers, will change as we support full OpenMP

- **While we have reduced overheads by 4x – 10x**
  - remaining overheads are due to the OpenMP standard
  - others are due to necessary locks / barriers / msyncs
  - compiler optimization can further reduce overheads in some cases

- **Barriers becoming the dominant factor at higher thread counts**

Part 2: Efficient Nested Parallelism

- **Examples of requests that are not currently possible**
  - get threads on separate cores to get more L1 cache
  - get threads collocated on same core to maximize cache reuse

- **Current runtimes have a fixed policy**
  - runtime tries to even out load balance across the machine
  - this works well for single level of parallelism,
  - not as well for nested parallelism

- **Want to allow users to specify where to get threads**
  - broad policies that cover most cases

- **Want to allow users to specify where threads are allowed to migrate**
  - for load balancing purpose

OpenMP Affinity Proposal

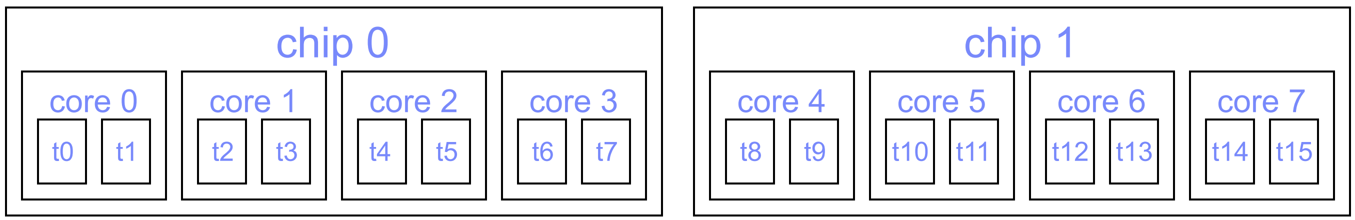
- **Define the concept of an OpenMP Place**
  - a set of one or more logical processors on which OpenMP-threads execute
  - OpenMP-threads may migrate within one place

- **Let the user specify its own set of places**
  - by default, the system defines its own list of places
  - in MPI hybrid mode, the “mpi-run” script would defines the set of places

- **Let the user specify how to recruit threads for OpenMP parallel**
  - MASTER: put threads in same place as master
  - CLOSE: put threads close to master
    - reduce false sharing, distribute among places
  - SPREAD: spread threads across the machine
    - reduce overheads of threads sharing the same core
    - optimize memory bandwidth by exploiting cores/sockets

How to use Place Lists

- **Consider a system with 2 chips, 4 cores, and 8 hardware-threads**



– One place per hardware-thread
  - OMP_PLACES=hwthread
  - OMP_PLACES=(0),(1),(2),…(15)

– One place per core, including both hardware-threads
  - OMP_PLACES=core
  - OMP_PLACES=(0,1),(2,3),(4,5)…(14,15)

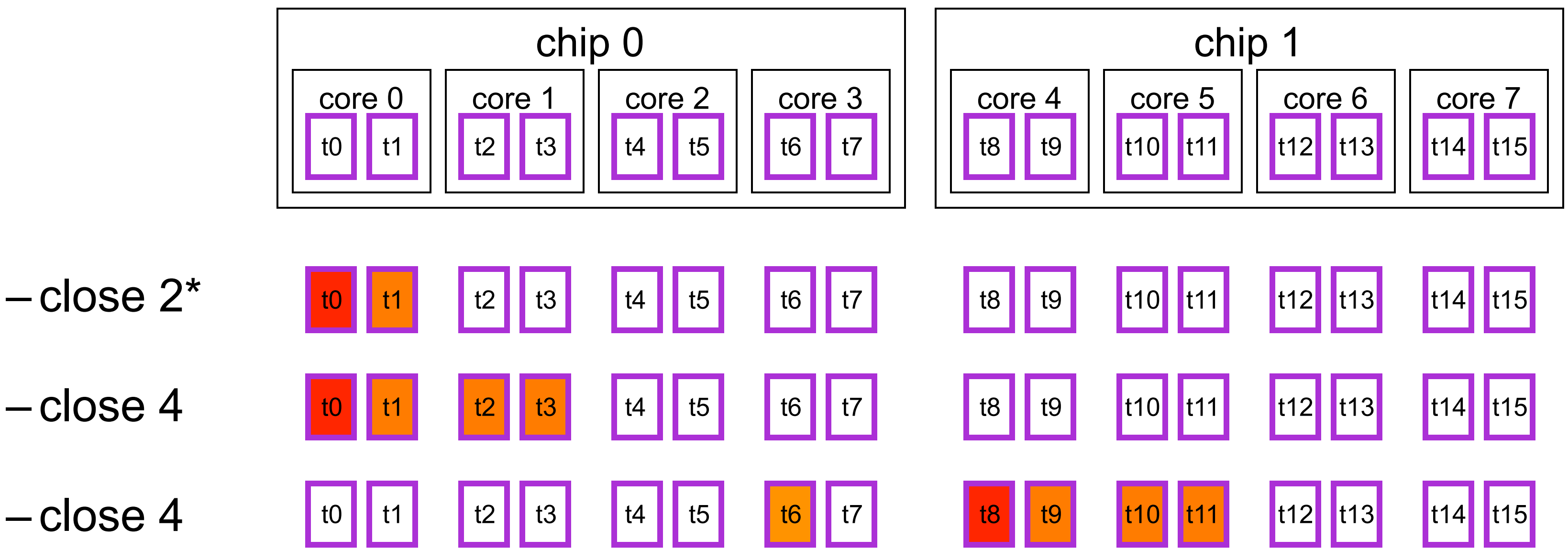
– One place per chip, excluding first hardware-thread
  - OMP_PLACES=(1,2,…,7),(9,10,11,..15)

CLOSE Policy

Compact selects OpenMP threads in the same place as the master

– consider the next place(s) when master place is full

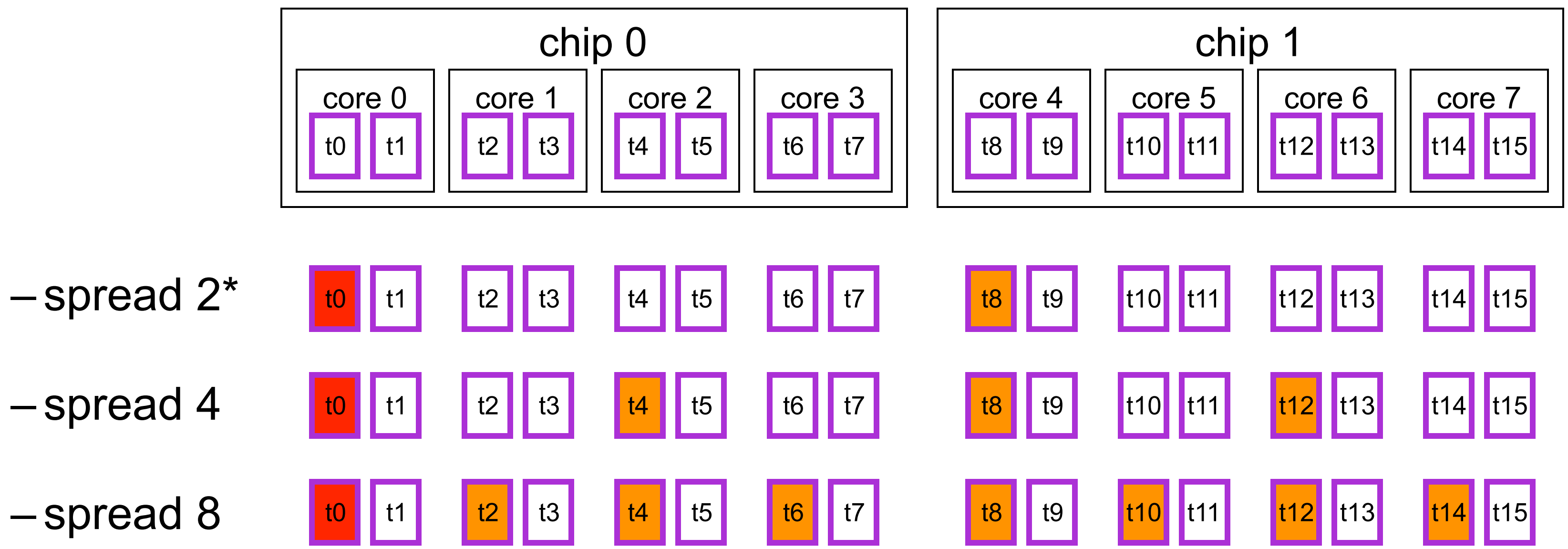
- **Example with OMP_PLACES=hwthread**



* technically "omp parallel num_threads(2) affinity(close)"   aster   orker

SPREAD Policy

- **Spread OpenMP threads as evenly as possible among places**
- **Example with OMP_PLACES=hwthread**

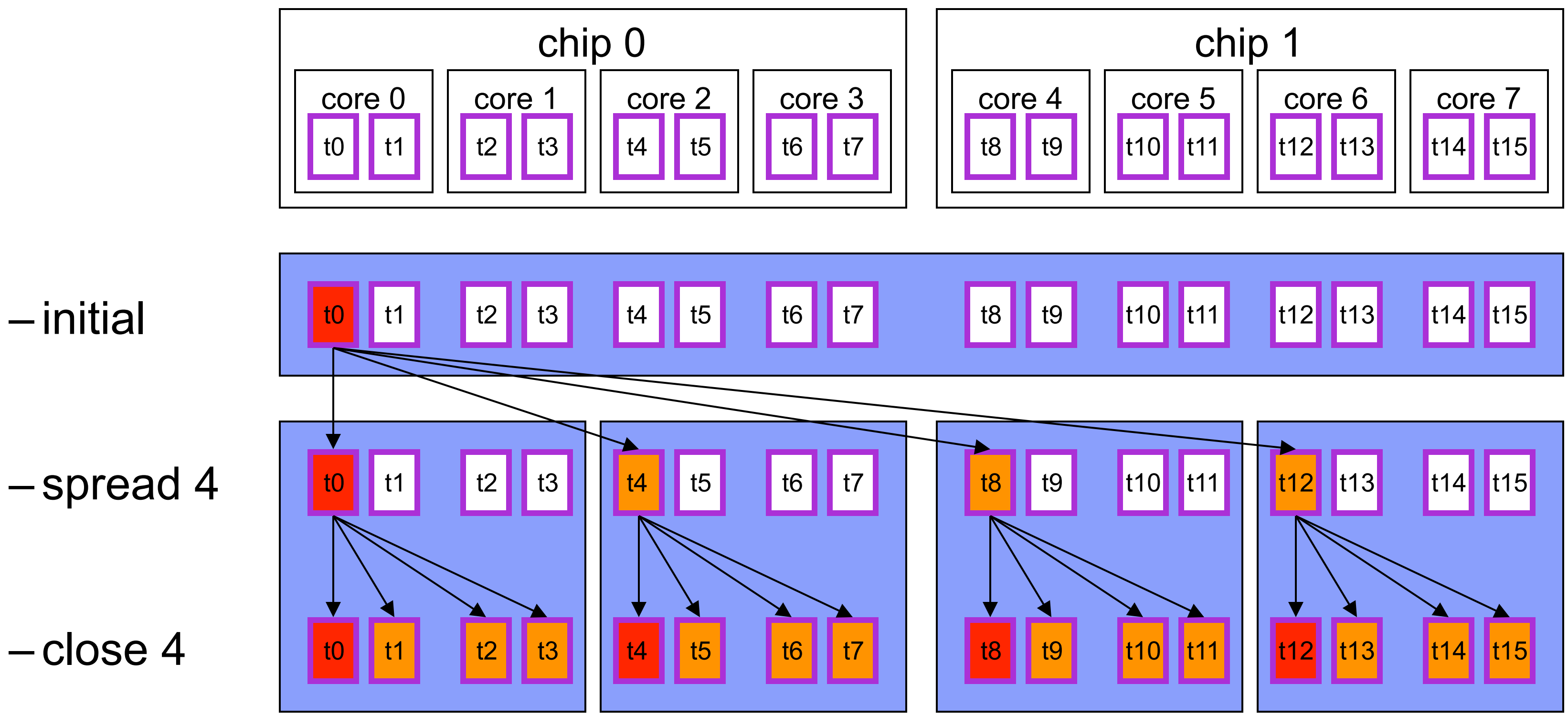


* technically "omp parallel num_threads(2) affinity(spread)"

Spread Policy Partition the Machine

- **Spread also implicitly partition the machine**
  - so that nested parallel-regions get threads only from its subset of the machine

- **Example: spread with nested, compact, parallel-regions**



partition

Observations

- **Give the user more fine-grain control**
  - which hardware thread / core / chip to use
  - which thread to select for a given parallel region
    - e.g. spread vs. compact
  - where threads are allowed to migrate (within a place)

- **Ongoing work**
  - implemented in our research OpenMP runtime
  - currently under review with the OpenMP Standard Language Committee

Part 3: Providing Timing Info

- **Possible approaches**
  - callbacks
  - statistical sampling (requires interrupt support)
  - embedded timing (using low overhead hardware timers)

- **Experimented with second approach**
  - approximate overheads: 100 cycles per OpenMP constructs
  - (for ref: 64-thread barrier 800-1000 cycles, parallel region 1800-2000 cycles)

- **Questions:**
  - what is needed by users
  - what is needed by tool developers
  - what can info can be provided cheaply

Timing Info: Cost Evaluation

- **Timing is relatively cheap on POWER**
  - get a local timer (register move)
  - save difference of 2 timer values (one store)

- **Saving a current state (idle-barrier/idle-lock)**
  - one store per transition

- **Callbacks**
  - load value of "enabled/disabled", one branch
  - BUT having a call has performance impact on optimized runtime
    - in optimized runtime, everything is inlined (except call outlined functions)
    - calls force caller-saved register back into memory (potentially 10+ load/store)
    - have seen overhead in 100+ cycles just for one additional function call
  - cheaper if are located just before/after outlined function calls