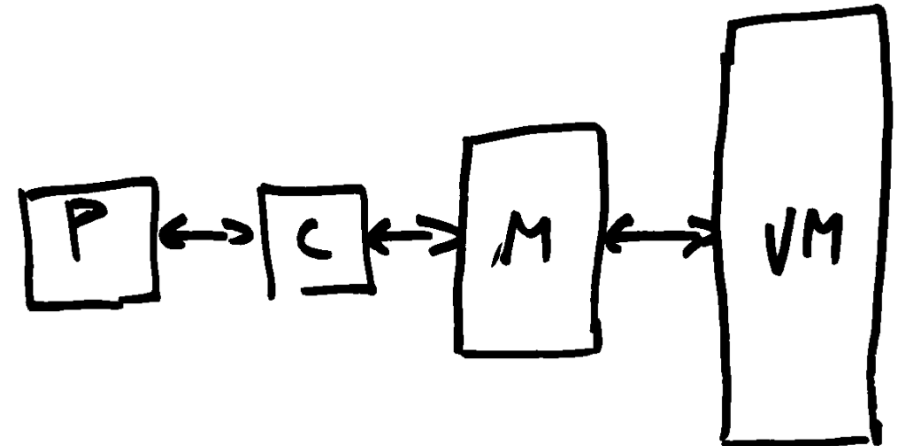
HPC:
Pitfalls of parallelism

Lack of locality

Temporal and spatial locality

Programs should entirely use the data pulled from the memory before moving to process another data

- Whole cache line is transferred from memory to cache
- Whole page is transferred from virtual memory to memory

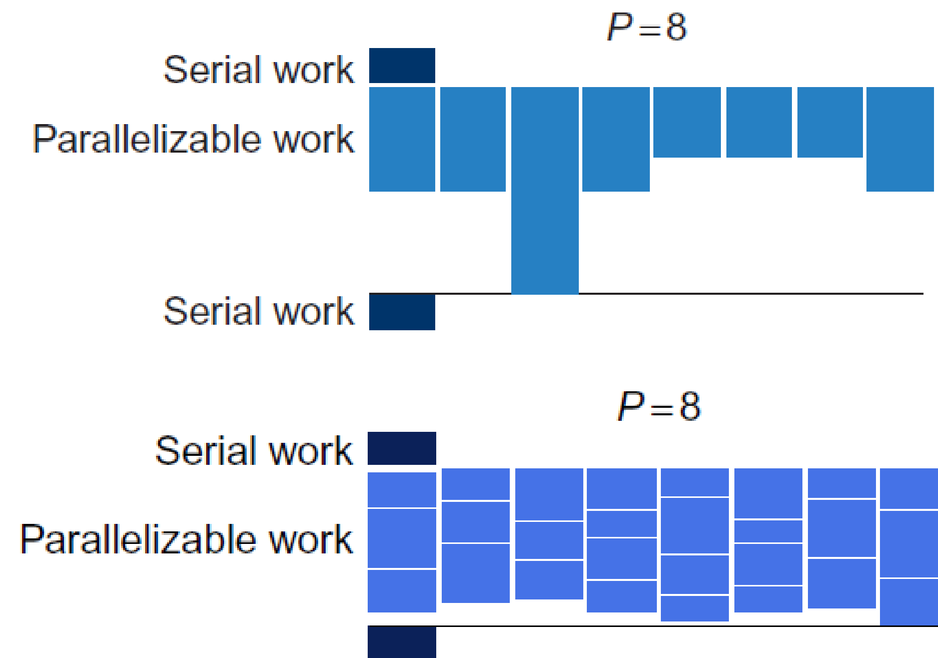


Load imbalance

Uneven distribution of work across workers

Prepare small chunks of work

Divide work to more tasks then there are workers

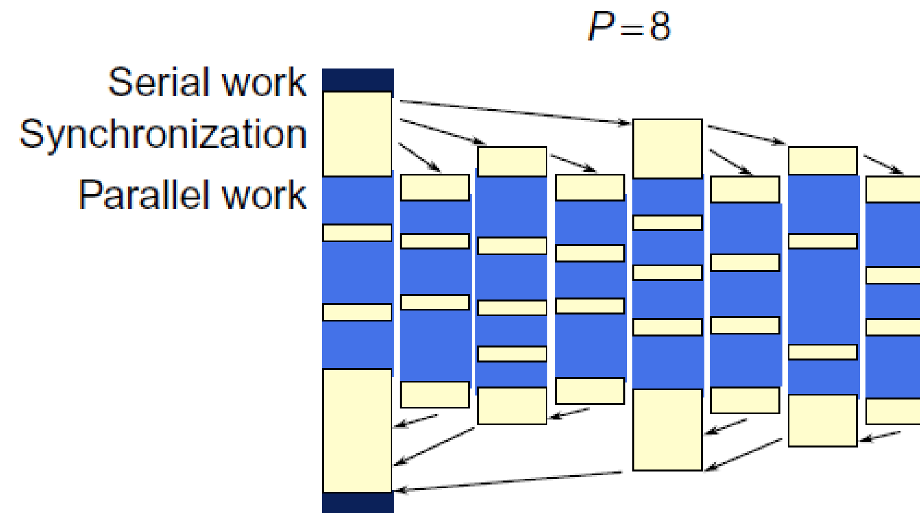


Synchronization

Dependencies between tasks require synchronization

Overhead

- Launching and synchronizing tasks
- Over-decomposition increases overhead



Race conditions

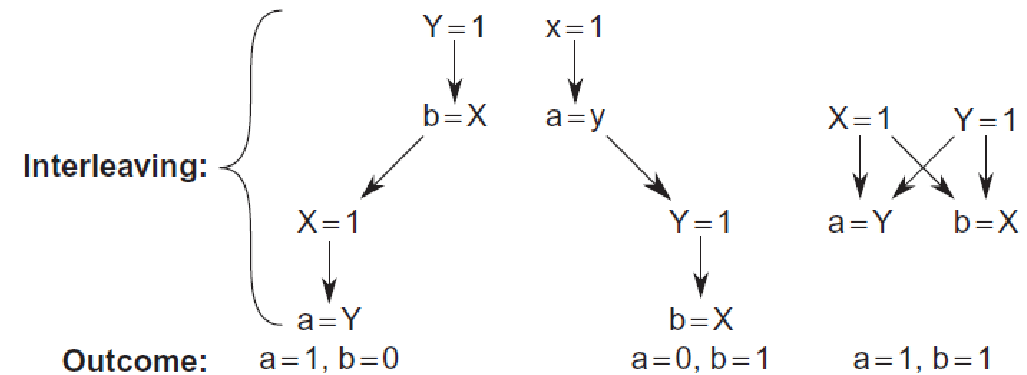
Concurrent tasks perform read and write operations at the same memory location

Are tricky, should be avoided

Example

- Initial values X = 0, Y = 0

Task A	Task B
X = 1;	Y = 1;
a = Y;	b = X;



Mutual exclusion and locks

Locks are low-level mechanism to avoid race conditions

Only one task is active at a time

They are expensive, should be the last choice!

Example

- Each task runs only once
- Possible changes in X without locks: +1, +2, +3, with locks: +3

Task A	Task B
`extern tbb::mutex M;`	`extern tbb::mutex M;`
`M.lock();`	`M.lock();`
`a = X;`	`b = X;`
`a += 1;`	`b += 2;`
`X = a;`	`X = b;`
`M.unlock();`	`M.unlock();`

Deadlock

At least two locks

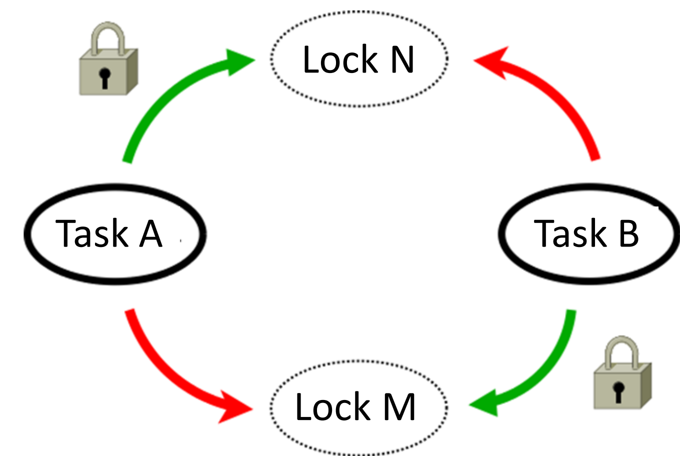
Two (or more) tasks wait for each other and cannot resume until the other task proceeds

Example

- Task A locks lock M and then tries to lock lock N
- Meanwhile task B locks lock N and then tries to lock lock M

Resolution

- Order of locking
- Lock and release



HPC: Shared memory systems

UROŠ LOTRIČ

Architecture

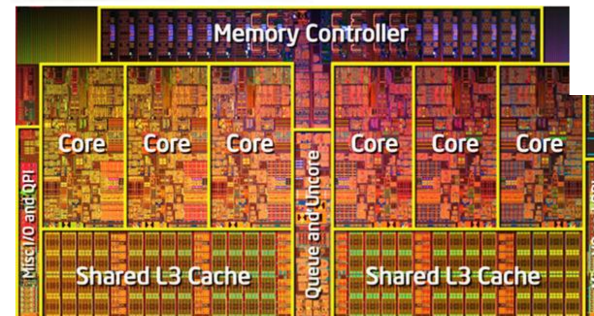
Multiple instructions, multiple data

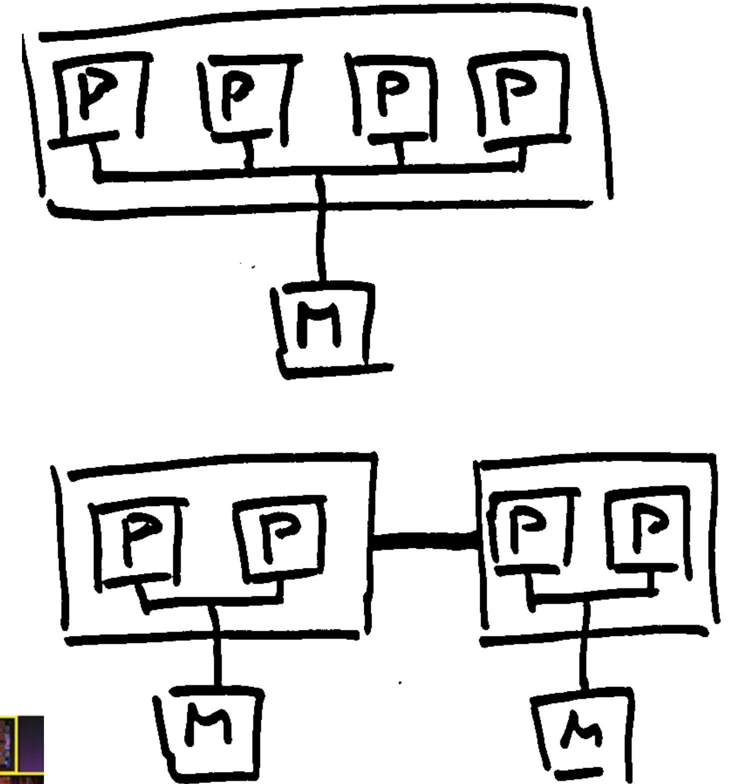
Share common memory

- Processors operate independently
- Processors can access each others' memory
- Changes in memory by one processor are visible to the others

UMA vs. NUMA

- Complexity and scalability
- Memory access times
- Cache coherence and false sharing may be magnified with NUMA
- UMA = SMP (Symmetric Multi Proc.)



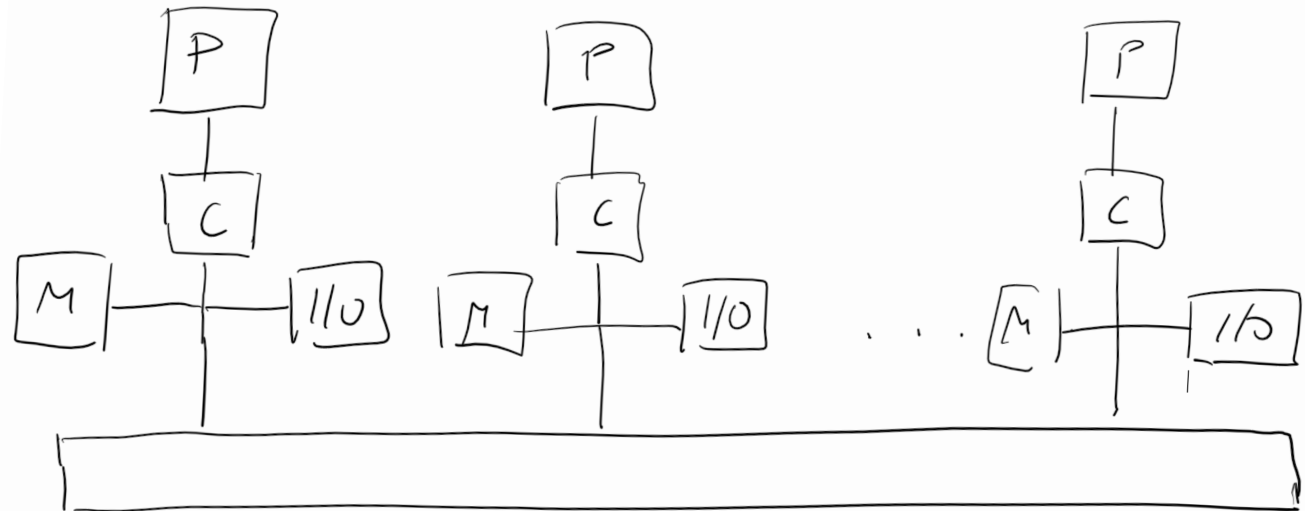


Architecture

Memory access

- Memory divided to many modules evenly distributed among processors
- Each core has its own cache

- Simple NUMA architecture



Data

Private

- used only by a single processor

Shared

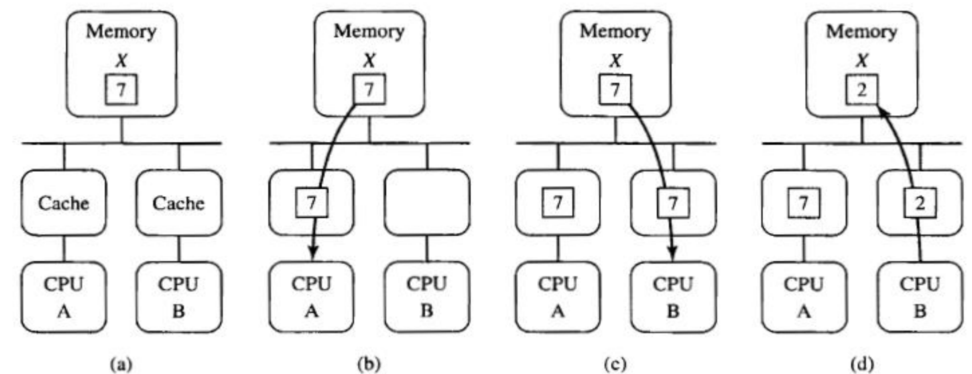
- used by multiple processors
- used for inter-process communication

Cache coherence

Different processors must see the same value at a given memory address

Snooping

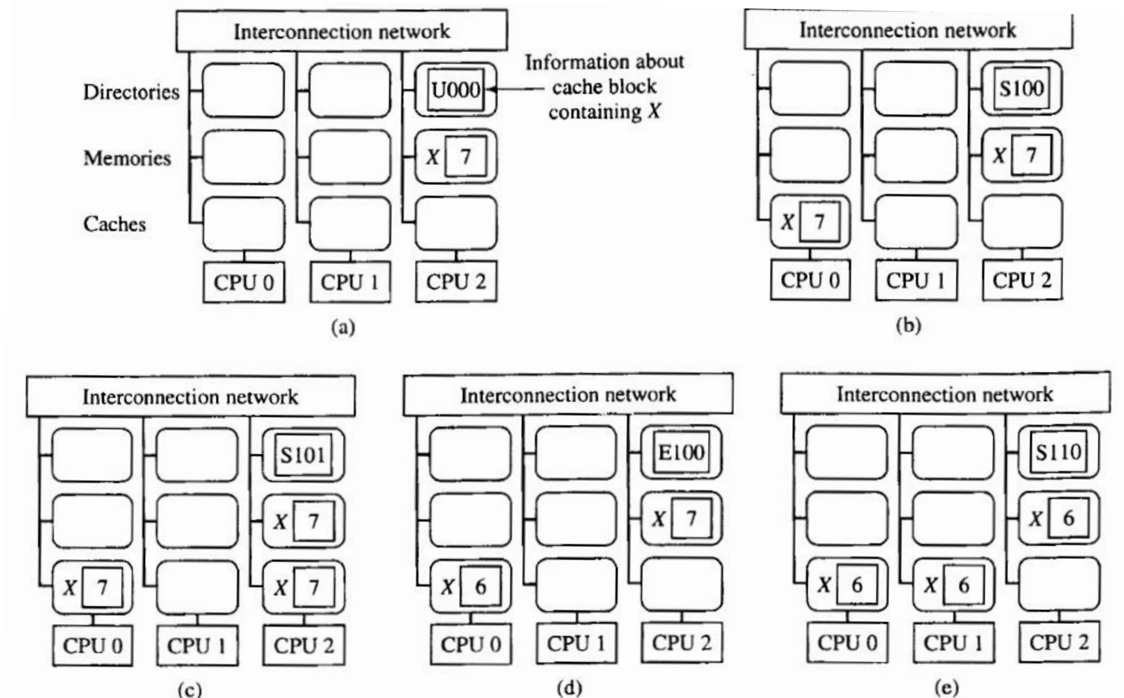
- Cache memory controller monitors bus to identify which blocks are written by other processors
- Write invalidate protocol (most common)
  - Get exclusive access before writing the data (write through)
  - Before writing all other copies with this data are invalidated
- Does not scale well



Cache coherence

Directory-based protocol

- Better suited for NUMA
- A single directory contains sharing information (statuses) about every block
  - To keep performance it is distributed among computer's local memories
- Must keep track which processors have copies to invalidate them when needed
- Simplified protocol
  - Statuses
    - U: uncached, not in any cache
    - S: shared by one or more processors, values in memory are correct,
    - E: exclusive, one processor has written the block, values in memory are obsolete
  - Illustration
    a. X has value 7, block is uncached
    b. State after CPU0 reads X
    c. State after CPU2 reads X
    d. State after CPU0 writes 6
    e. State after CPU1 reads 6



False sharing

Cache is implemented in hardware, it operates on cache lines

Even if processors do not operate on the same variable but on variables located in the same cache line, the cache coherence protocols are involved

Does not produce false results

Disastrous consequences on performance

Processor synchronization

Performed over shared variables

Prevents false computation

Types

- Mutual exclusion: only one processor can be engaged in a specific activity at any time
- Barrier synchronization: all processors must reach the barrier before any processor can proceed

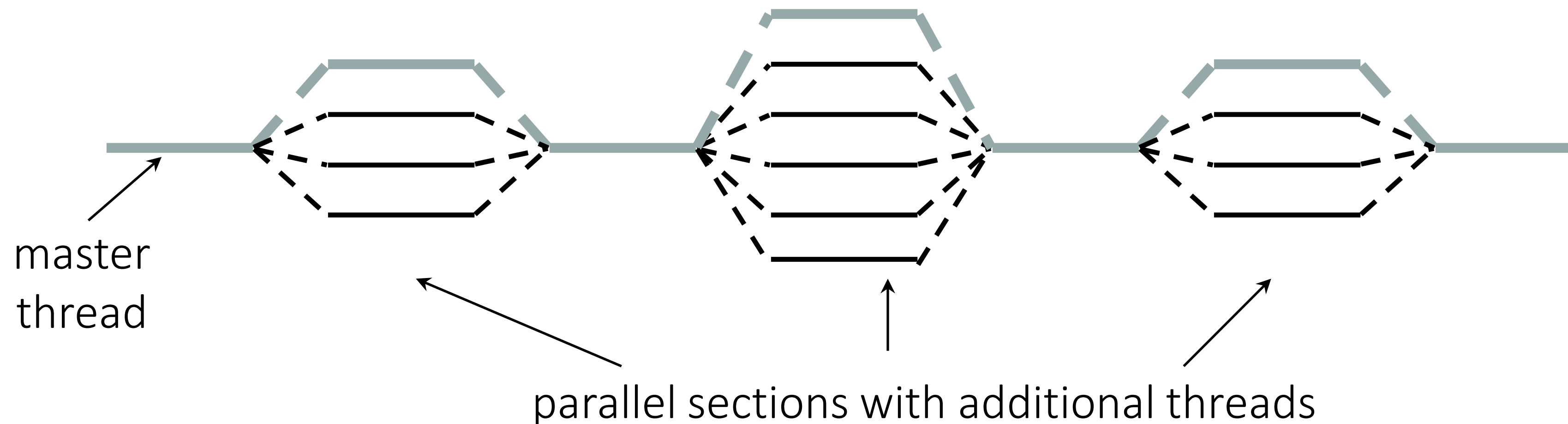
Threads

When program starts only the main thread (master thread) is active

Where a parallel operation is required, master thread creates additional threads

Threads work concurrently

At the end of a parallel operation, additional threads are suspended, and only master thread continues

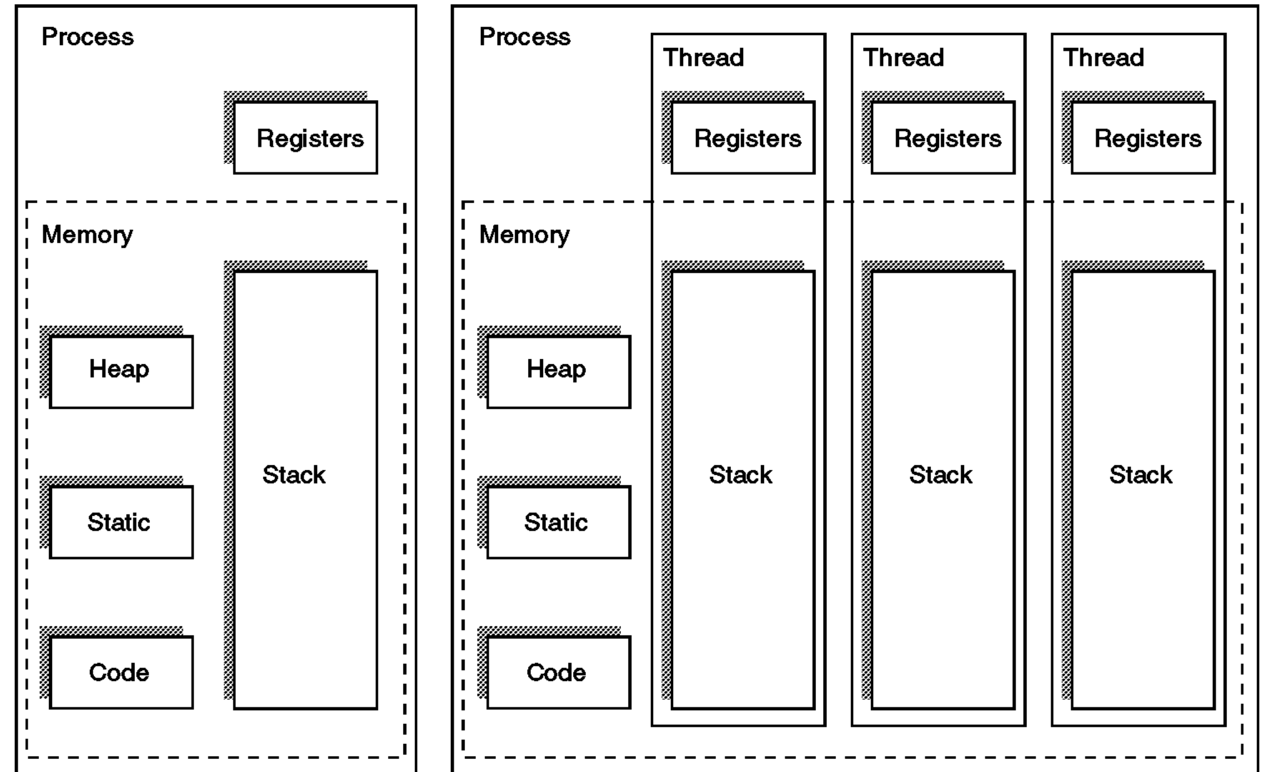


Threads

Single threaded and multi threaded processes →

Each thread has its own
- Program counter
- Stack and stack pointer
- registers

Threads share
- Code segment
- Heap
- Static variables



OpenMP

OpenMP

Open Multi**P**rocessing

API for shared-memory parallel programming

Usage

- Compiler directives #pragma omp ...
- Functions
- Environment variables

Compiling

- #include "omp.h"
- gcc ... -f openmp

References

- https://riptutorial.com/openmp

Important compiler directives

parallel

- issues many threads which execute in parallel

parallel for

- divides independent iterations among threads

parallel sections

- divides sections of code to execute in parallel

critical, atomic

- ensures that only a single thread enters a section
- atomic is faster, but only for limited scope of commands

single, master

- only one thread (master) executes a part of a code

Useful functions

Number of threads

- omp_set_num_threads: sets number of threads to be used in parallel section
- omp_get_num_threads: get currently set number of threads
- omp_get_max_threads: get max. number of threads (on hardware level)
- omp_get_thread_num: get thread index 0 … omp_get_num_threads

Number of processors/cores

- omp_num_procs: number of processors available

Timing

- omp_get_wtime: wall-clock time elapsed from some point in the past
- omp_get_wtick: resolution of wall-clock time measurement

Environment variables

User can influence on behaviour of executable code at runtime

Set number of threads

- set OMP_NUM_THREADS=4
- export OMP_NUM_THREADS=4

Distribution of work

parallel

```
#include ”omp.h”
void main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(“hello(%d) ”, ID);
        printf(“world(%d) \n”, ID);
    }
}
```

```
Output

hello(1) hello(0)
world(1) world(0)
hello(3) hello(2)
world(3) world(2)
```

Distribution of work

section, parallel

- barrier is automatically set after each sections directive
- to disable it add nowait after sections

```
#include "omp.h"
#include <stdio.h>

void main(void)
{
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            {
            int ID = omp_get_thread_num();
            printf("hello(%d) ", ID);
            }
#pragma omp section
            {
            int ID = omp_get_thread_num();
            printf("world(%d) ", ID);
            }
        }
    }
}
```

False sharing example

Opteron processor

- Cache line size 64B = 8 x 8B

Length of double is 8B

Results based on distance dist between counters in the array

- 1 … 7: execution time around 50s,
- 8 and above: execution time around 10s
- Number of counts is always correct

When counters in separate cache lines the performance is severely increased

```
int main(int argc, char *argv[])
{
    double *counter;
    int n = 2000000000;
    int i, dist;
    double dt;

    dist = atoi(argv[1]);
    counter = (double *)calloc(sizeof(double), dist+1);

    omp_set_num_threads(2);
    dt = omp_get_wtime();
    #pragma omp parallel private(i)
    {
        int id = omp_get_thread_num()*dist;
        for(i = 0; i < n; i++)
            counter[id]++;
    }
    dt = omp_get_wtime() - dt;

    printf("equal = %d\n", counter[0] == counter[dist]);
    printf("time = %lf\n\n", dt);

    free(counter);

    return 0;
}
```