An Introduction into Parallelization

Part II - Design

Jascha Schewtschenko

Royal Observatory of Edinburgh, University of Edinburgh

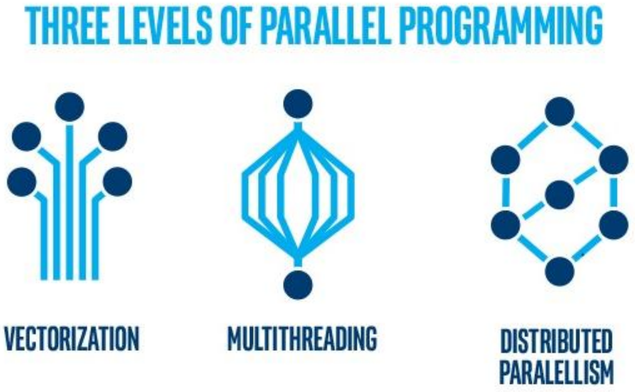
May 16, 2025

Outline

1. Parallel Programming Models (cont.)
   - Recap: Computer Architectures
   - Distributed Parallelism with Message Passing
   - Hybrid Models
2. Designing Parallel Programs
   - Understand your problem and tools
   - Partitioning - Domain vs functional decomposition
   - Data Dependence / Race conditions
   - Synchronization
   - Communication
   - Load balancing
   - I/O
   - Performance Analysis & Tuning
   - Conclusions / tl;dr

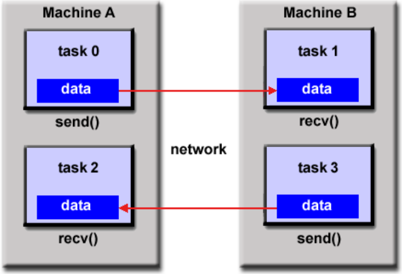
Parallel Programming Models

Parallel Programming Models - Overview



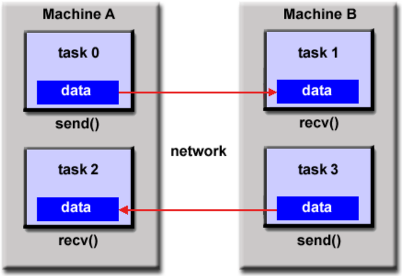
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.



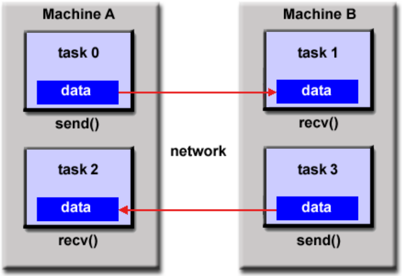
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")



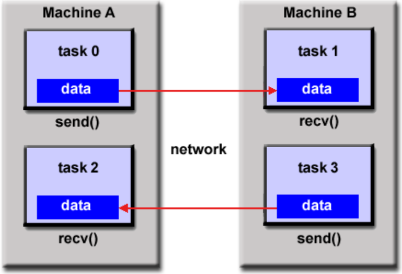
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)



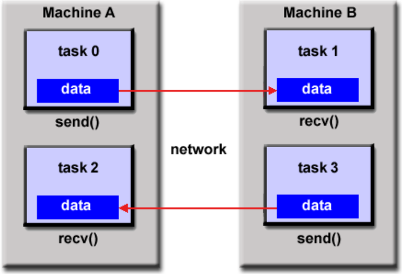
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)
- The message system can also be used to synchronize processes



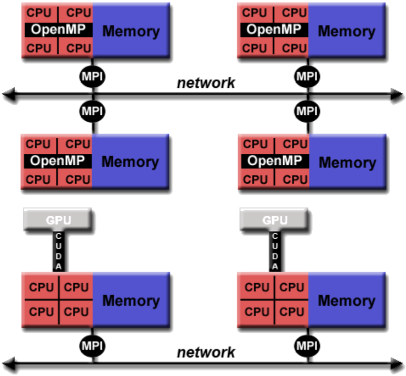
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)
- The message system can also be used to synchronize processes
- *MPI* is the "de facto" standard



Hybrid Models

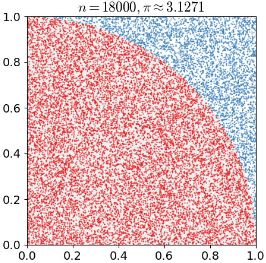
- Allows to make best use of locally shared memory or hardware, while still allowing for a good scalability across multiple nodes
- Comes with a significant increase in complexity/costs
- certain incompatibilities between libraries may exist (e.g. lack of thread-safety of MPI library)



Designing Parallel Programs

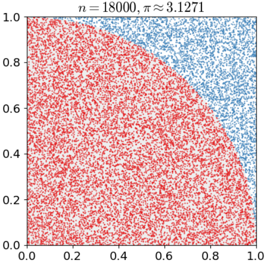
Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling

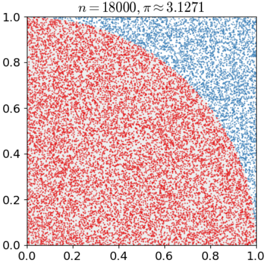


- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



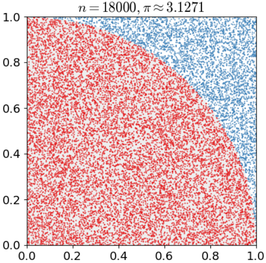
- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

- Identify inhibitors to parallelism: data-dependencies, I/O bottlenecks

Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

- Identify inhibitors to parallelism: data-dependencies, I/O bottlenecks
- Consider replacing your algorithms with equivalent ones better suited for parallelism

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure
- make use of hardware optimization (e.g. vectorization, optimized libraries like MKL)

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure
- make use of hardware optimization (e.g. vectorization, optimized libraries like MKL)
- identify hotspots in your program, i.e. routines where program spends lots of time in and check for improvement in parallelism ($\rightarrow$Amdahl's Law/Scaling)

Partitioning - Domain decomposition

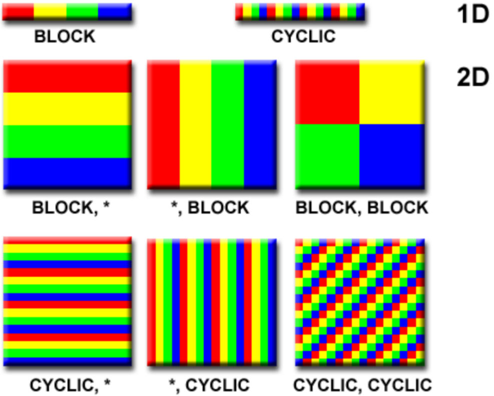
- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.

Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:

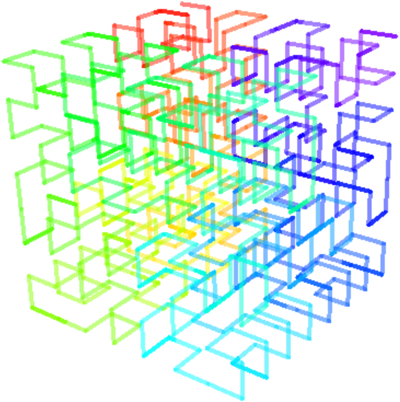
Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:



Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:

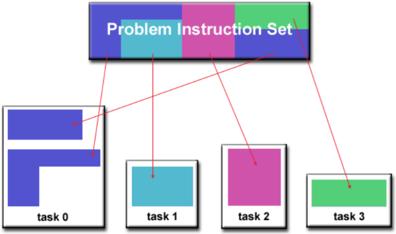


Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.

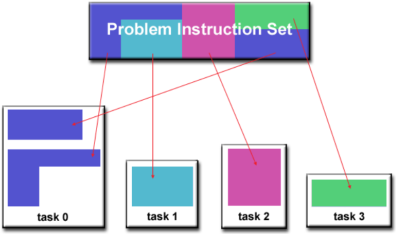
Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Tasks/threads then specialize in doing specific parts of the overall work:



Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Tasks/threads then specialize in doing specific parts of the overall work:



- implemented e.g. in master/slave paradigm (see exercises)

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism (cf. Fibonacci series).

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism (cf. Fibonacci series).
- This can also cause a so called race condition:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Synchronization

- synchronization is used to control the flow of processes/threads to deal with "mutually exclusive" resources (using locks/semaphores) to resolve e.g. race conditions

Synchronization

- synchronization is used to control the flow of processes/threads to deal with “mutually exclusive” resources (using locks/semaphores) to resolve e.g. race conditions
- ... or to synchronize the calculations of processes (using barriers) for communication to exchange results or to redistribute the workload

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
    - Communication overhead

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth

Communication

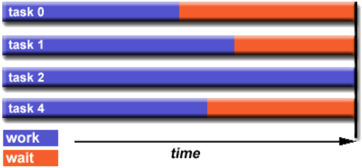
- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth
  - Synchronous vs. asynchronous communications

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth
  - Synchronous vs. asynchronous communications
  - Point-to-Point vs. collective communications

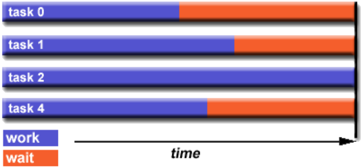
Load balancing

- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



Load balancing

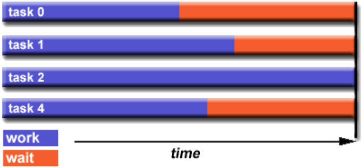
- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



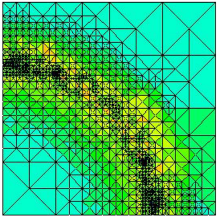
- requires well-balanced workload distribution between processors

Load balancing

- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



- requires well-balanced workload distribution between processors
- difficult in heterogeneous, dynamic problem sets with incomplete information about the actual workload

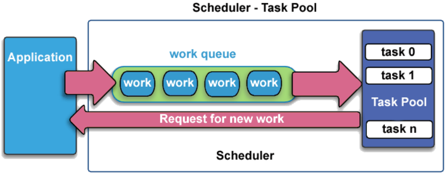


Load balancing (cont.)

- requires repeated repartitioning the problem based on estimate of workload

Load balancing (cont.)

- requires repeated repartitioning the problem based on estimate of workload
- alternatively, use asynchronous approach with scheduler-task pool with smaller workload packages



I/O

- data transfer to storage medias, network file servers

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system
- Parallel filesystems available (e.g. GPFS, Lustre, HDFS)

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system
- Parallel filesystems available (e.g. GPFS, Lustre, HDFS)
- more on this in the "HPC & (Big) Data" talk!

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer, ARM DDT

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer, ARM DDT
- Unfortunately, covering this topic in all the detail would go beyond the scope of this introduction to parallel program, so we focus on a simple python profiler as an example, `Yappi`.

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer, ARM DDT
- Unfortunately, covering this topic in all the detail would go beyond the scope of this introduction to parallel program, so we focus on a simple python profiler as an example, `Yappi`.

Two caveats:

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer, ARM DDT
- Unfortunately, covering this topic in all the detail would go beyond the scope of this introduction to parallel program, so we focus on a simple python profiler as an example, `Yappi`.

Two caveats:

- **Heisenberg's uncertainty principle**

  > *The more precisely you know the position of something, the less precisely you can tell its speed and momentum.*

  applies to profiling, too!

Performance Analysis & Tuning

- Analysing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer, ARM DDT
- Unfortunately, covering this topic in all the detail would go beyond the scope of this introduction to parallel program, so we focus on a simple python profiler as an example, `Yappi`.

Two caveats:

- **Heisenberg's uncertainty principle**

  > *The more precisely you know the position of something, the less precisely you can tell its speed and momentum.*

  applies to profiling, too!
- profilers do **NOT** tell you **HOW** to optimize your code, they merely tell you **WHERE** to (potentially) do so for the best yield.

Performance Analysis & Tuning - `Yappi`

- **Y**(et)**A**(nother)**P**(ython)**P**(rof)**I**(ler) supports multi-threaded python out of the box; allows for easy CPU-time measurements

Performance Analysis & Tuning - `Yappi`

- **Y**(et)**A**(nother)**P**(ython)**P**(rof)**I**(ler) supports multi-threaded python out of the box; allows for easy CPU-time measurements
- (multi)processes have to be profiled seperately (cf. Exercises)

Performance Analysis & Tuning - `Yappi`

- **Y**(et)**A**(nother)**P**(ython)**P**(rof)**I**(ler) supports multi-threaded python out of the box; allows for easy CPU-time measurements
- (multi)processes have to be profiled seperately (cf. Exercises)
- exports results into callgrind & pstat format (to be loaded by analyzers/visualisation tools)

```
Function stats for (_MainThread) (0)

Clock type: WALL
Ordered by: totaltime, desc

name                                  ncall  tsub      ttot      tavg
..hon3.8/threading.py:859 Thread.run  2      0.000017  44.09970  22.04985
..eads_revisited/helper.py:26 runall  2      0.002906  44.09969  22.04984
time_trials_threading.py:11 worker    10     0.052127  44.09676  4.409676
..ads_revisited/helper.py:10 foreach  1      0.000022  22.06303  22.06303
..on3.8/threading.py:979 Thread.join  2      0.000016  22.06242  11.03121
..:1017 Thread._wait_for_tstate_lock  2      0.000014  22.06240  11.03120
time_trials_threading.py:23 g         10     0.084264  9.152582  0.915258
..._trials_threading.py:24 <listcomp> 10     7.603336  9.068318  0.906832
time_trials_threading.py:26 h         10     7.371068  8.857567  0.885757
time_trials_threading.py:19 f         10     5.973178  5.973178  0.597318
..ges/numpy/core/numeric.py:144 ones  10     0.000200  0.045074  0.004507
..ray_function__ internals>:2 copyto  10     0.000135  0.044414  0.004441
..n3.8/threading.py:834 Thread.start  2      0.000023  0.000501  0.000250
..hon3.8/threading.py:540 Event.wait  2      0.000014  0.000375  0.000188
...8/threading.py:270 Condition.wait  2      0.000027  0.000350  0.000175
..._revisited/helper.py:52 <listcomp> 1      0.000011  0.000091  0.000091
..8/threading.py:761 Thread.__init__  2      0.000039  0.000080  0.000040
...8/threading.py:505 Event.__init__  2      0.000006  0.000021  0.000010
..mpy/core/multiarray.py:1043 copyto  10     0.000018  0.000018  0.000002
..n3.8/threading.py:944 Thread._stop  2      0.000014  0.000016  0.000008
..hreading.py:222 Condition.__init__  2      0.000014  0.000014  0.000007
..reading.py:261 Condition._is_owned  2      0.000005  0.000009  0.000005
..8/threading.py:1306 current_thread  4      0.000008  0.000009  0.000002
..n3.8/_weakrefset.py:81 WeakSet.add  2      0.000007  0.000008  0.000004
..reading.py:246 Condition.__enter__  2      0.000004  0.000006  0.000003
..ing.py:255 Condition._release_save  2      0.000004  0.000005  0.000002
..hreading.py:249 Condition.__exit__  2      0.000005  0.000005  0.000002
..ython3.8/_weakrefset.py:38 _remove  2      0.000004  0.000004  0.000002
...py:258 Condition._acquire_restore  2      0.000001  0.000003  0.000002
..ython3.8/threading.py:734 _newname  2      0.000003  0.000003  0.000002
..ng.py:1177 _make_invoke_excepthook  2      0.000003  0.000003  0.000002
..reading.py:1095 _MainThread.daemon  4      0.000002  0.000002  0.000000
..n3.8/threading.py:513 Event.is_set  4      0.000001  0.000001  0.000000
```

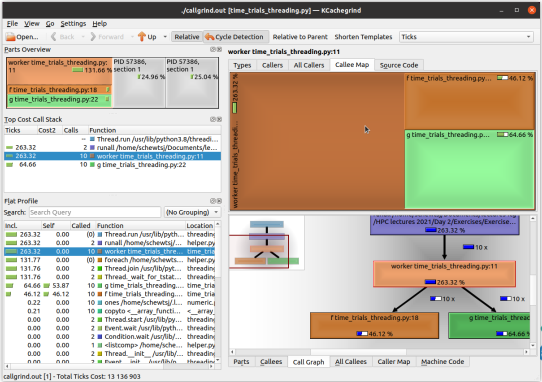
Performance Analysis & Tuning - Yappi

- **Y**(et)**A**(nother)**P**(ython)**P**(rof)**I**(ler) supports multi-threaded python out of the box; allows for easy CPU-time measurements
- (multi)processes have to be profiled seperately (cf. Exercises)
- exports results into callgrind & pstat format (to be loaded by analyzers/visualisation tools)

```
Function stats for (_MainThread) (0)

Clock type: WALL
Ordered by: totaltime, desc

name                                ncall  tsub      ttot      tavg
..hon3.8/threading.py:859 Thread.run  2      0.000017  44.09970  22.04985
..eads_revisited/helper.py:26 runall  2      0.002906  44.09969  22.04984
time_trials_threading.py:11 worker    10     0.052127  44.09676  4.409676
..ads_revisited/helper.py:10 foreach  1      0.000022  22.06303  22.06303
..on3.8/threading.py:979 Thread.join  2      0.000016  22.06242  11.03121
..:1017 Thread._wait_for_tstate_lock  2      0.000014  22.06240  11.03120
time_trials_threading.py:23 g         10     0.084264  9.152582  0.915258
.._trials_threading.py:24 <listcomp>  10     7.603336  9.068318  0.906832
time_trials_threading.py:26 h         10     7.371068  8.857567  0.885757
time_trials_threading.py:19 f         10     5.973178  5.973178  0.597318
..ges/numpy/core/numeric.py:144 ones  10     0.000200  0.045074  0.004507
..ray_function__ internals>:2 copyto  10     0.000135  0.044414  0.004441
..n3.8/threading.py:834 Thread.start  2      0.000023  0.000501  0.000250
..hon3.8/threading.py:540 Event.wait  2      0.000014  0.000375  0.000188
...8/threading.py:270 Condition.wait  2      0.000027  0.000350  0.000175
.._revisited/helper.py:52 <listcomp>  1      0.000011  0.000091  0.000091
..8/threading.py:761 Thread.__init__  2      0.000039  0.000080  0.000040
...8/threading.py:505 Event.__init__  2      0.000006  0.000021  0.000010
..mpy/core/multiarray.py:1043 copyto  10     0.000018  0.000018  0.000002
..n3.8/threading.py:944 Thread._stop  2      0.000014  0.000016  0.000008
..hreading.py:222 Condition.__init__  2      0.000014  0.000014  0.000007
..reading.py:261 Condition._is_owned  2      0.000005  0.000009  0.000005
..8/threading.py:1306 current_thread  4      0.000008  0.000009  0.000002
..n3.8/_weakrefset.py:81 WeakSet.add  2      0.000007  0.000008  0.000004
..reading.py:246 Condition.__enter__  2      0.000004  0.000006  0.000003
..ing.py:255 Condition._release_save  2      0.000004  0.000005  0.000002
..hreading.py:249 Condition.__exit__  2      0.000005  0.000005  0.000002
..ython3.8/_weakrefset.py:38 _remove  2      0.000004  0.000004  0.000002
...py:258 Condition._acquire_restore  2      0.000001  0.000003  0.000002
..ython3.8/threading.py:734 _newname  2      0.000003  0.000003  0.000002
..ng.py:1177 _make_invoke_excepthook  2      0.000003  0.000003  0.000002
..reading.py:1095 _MainThread.daemon  4      0.000002  0.000002  0.000000
..n3.8/threading.py:513 Event.is_set  4      0.000001  0.000001  0.000000
```



Conclusions / tl;dr

- Writing code that does the job is one thing; writing it in an optimized way is a different game altogether ⇒ HPC/HTC requires highly optimized code for “best” performance!

Conclusions / tl;dr

- Writing code that does the job is one thing; writing it in an optimized way is a different game altogether ⇒ HPC/HTC requires highly optimized code for "best" performance!
- Parallelization techniques provide ways to reduce the "walltime" of a code (usually at the cost of an increased "CPU time"); requires in-depth knowledge in parallel algorithms

Conclusions / tl;dr

- Writing code that does the job is one thing; writing it in an optimized way is a different game altogether ⇒ HPC/HTC requires highly optimized code for “best” performance!
- Parallelization techniques provide ways to reduce the “walltime” of a code (usually at the cost of an increased “CPU time”); requires in-depth knowledge in parallel algorithms
- often, there are already ”low-hanging fruits” in the serial code (reduction of I/O, amount of function calls, object conversions, etc.); demands in-depth knowledge of how things work ”under the hood”.

Conclusions / tl;dr

- Writing code that does the job is one thing; writing it in an optimized way is a different game altogether $\Rightarrow$ HPC/HTC requires highly optimized code for “best” performance!
- Parallelization techniques provide ways to reduce the “walltime” of a code (usually at the cost of an increased “CPU time”); requires in-depth knowledge in parallel algorithms
- often, there are already "low-hanging fruits" in the serial code (reduction of I/O, amount of function calls, object conversions, etc.); demands in-depth knowledge of how things work "under the hood".
- tools like profilers can help you to identify what to optimize, but you still need the knowledge on how to do this.