Parallelism
1202

Second deliverable

Héctor Ramón Jiménez

Alvaro Espuña Buxo

April 3, 2014
Facultat d'Informàtica de Barcelona

Parallelization overheads

1. **Which is the overhead associated with the activation of a `parallel` region in OpenMP? Is it constant? Reason the answer based on the results reported by the `ompparallel` code and the the trace visualized with Paraver.**

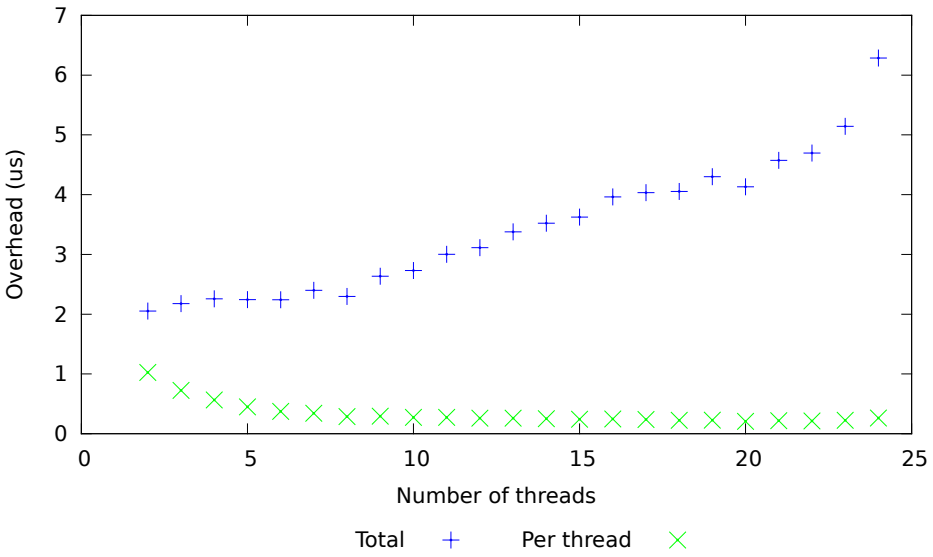

Figure 1: Overheads of the activation of parallel regions

The figure above shows that the activation of the `parallel` region has a total overhead that **grows with the number of threads**. It shows that the overhead per thread remains constant ($\approx 0.333\mu s$), which means that the total overhead grows linearly.
Paraver shows clearly how the main thread needs to create every other thread **independently** and, at the end, join and terminate all of them. Thus, the time of the creation and termination depends on the number of threads to create/terminate.

2. **Which is the minimum overhead associated with the execution of critical regions in OpenMP?**
The minimum overhead occurs when a single thread is used. When there is only one thread it can `lock` the region at any moment without waiting. However, `lock` and `unlock` take time even if there are no other threads involved. This means that critical regions with one thread have an overhead compared to not using them.
To calculate the minimum overhead per critical region executed we can use the time (in ns) of `dotprod_serial` and `dotprod_mutex_everytime`

with one single thread:

$$T_{critical_1} = \frac{T_{1\_mutex\_everytime} - T_{1\_serial}}{N_{critical\_regions}} = \frac{1.079 - 0.315}{2^{26}} \cdot 10^9 = 11.384ns$$

3. **In the presence of lock conflicts and true sharing (as it happens in dotprod mutex everytime), how the overhead associated with critical increases with the number of processors? How this overhead is decomposed? Reason the answer based on the results visualized with Paraver.**

Number of threads	**Time (s)**
1	1
2	7
4	24
8	35

As the table above shows, the overhead grows far from a linear way. The chances of a successful `lock` decrease as the number of threads grows. Paraver shows that every thread spends approximately, when using 4 or 8 threads, a **60%** of the execution time in the `lock` phase when only stays `locked` a **1%** of it.
We can calculate the overhead associated with the critical regions in function of the number of threads. If we use P threads then the work performed by `dotprod_serial` is shared equitatively. We have to add to the equation the overhead of creation/termination of the additional threads:

$$T_{critical_P} = \frac{T_{P_{mutex\_everytime}} - \frac{T_{P_{serial}}}{P} - 0.333 \cdot (P-1)}{N_{critical\_regions}}$$

For example, with $P = 8$:

$$T_{critical_8} = \frac{35.447 - \frac{0.315}{8} - 0.333 \cdot 7}{2^{26}} \cdot 10^9 = 492.880ns$$

This is actually $\frac{T_{critical_8}}{T_{critical_1}} - 1 = 42.294$ times more overhead than using one thread!

4. **In the presence of `false sharing` (as it happens in dotprod vectorsum), which is the additional average access latency that you observe to memory? Which causes this increase in the memory access time? Reason the answer based on the results visualized with Paraver.**
   The additional average access latency to memory can be calculated using the time of the versions with and without `false sharing`:

$$T_m = \frac{T_{no\_padding} - T_{padding}}{\frac{N_{accesses}}{P}} = \frac{0.243 - 0.069}{\frac{2^{26}}{8}} \cdot 10^9 = 20.742416ns$$

   These are the Paraver histograms that show noticeable differences:

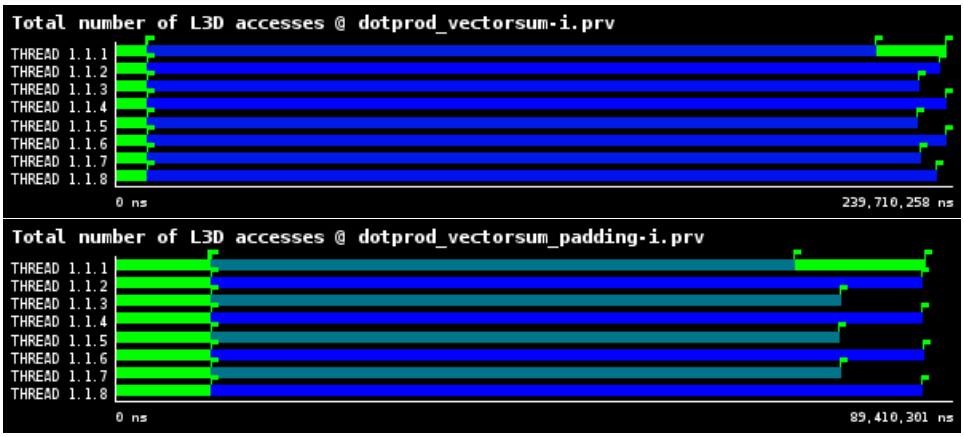

Figure 2: Differences in level 3 cache accesses

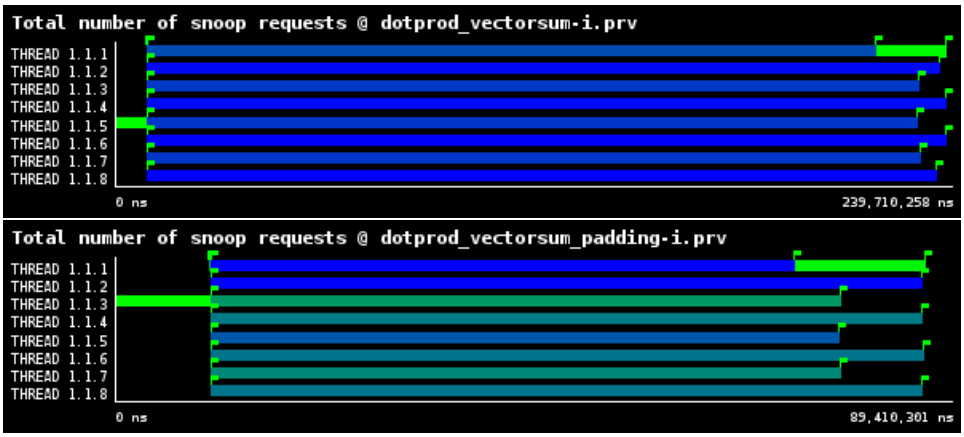

Figure 3: Differences in snoop requests

The histograms show that `false sharing` produces a huge increase in the accesses to level 3 caches and snoop requests. This means that

the first two levels of caches have the data `invalidated` because some other thread has modified **part of it**. That's the cause of the increase in memory access time.

Execution time and speedup

5. **Complete the following table with the execution times of the different versions of dotprod that we provide to you. The speed–up has to be computed with respect to the execution of the serial version. For each version and number of threads, how many executions have you performed?**

version	**1 processor**	**8 processors**	**speed–up**
dotprod_serial	0.315	-	1
dotprod_mutex_everytime	1.079	35.447	0.292
dotprod_mutex	0.360	0.067	4.701
dotprod_vectorsum	0.383	0.242	1.302
dotprod_vectorsum_padding	0.385	0.069	4.565

Two executions for each version and number of threads.