PAR — Deliverable for Laboratory Session 1

Víctor Jiménez Arador, Miguel Moreno Gómez

Group 12 — Fall semester 2017/2018

1 Node architecture and memory

1. Draw and briefly describe the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).

1.1 CPU data (`lscpu`)

- **Number of sockets:** 6
- **Number of cores per socket:** 6
- **Number of threads per core:** 2

1.2 Cache hierarchy and memory (`lstopo`)

- **Level 1 data and instructions cache:** 32 KB (each)
- **Level 2 cache:** 256 KB
- **Level 3 cache:** 12288 KB

1.3 Main memory

- **Memory:** 24 GB (2x12 GB)
- **Available memory:** 23281756 KB
- **Total amount of memory:** 24625048 KB

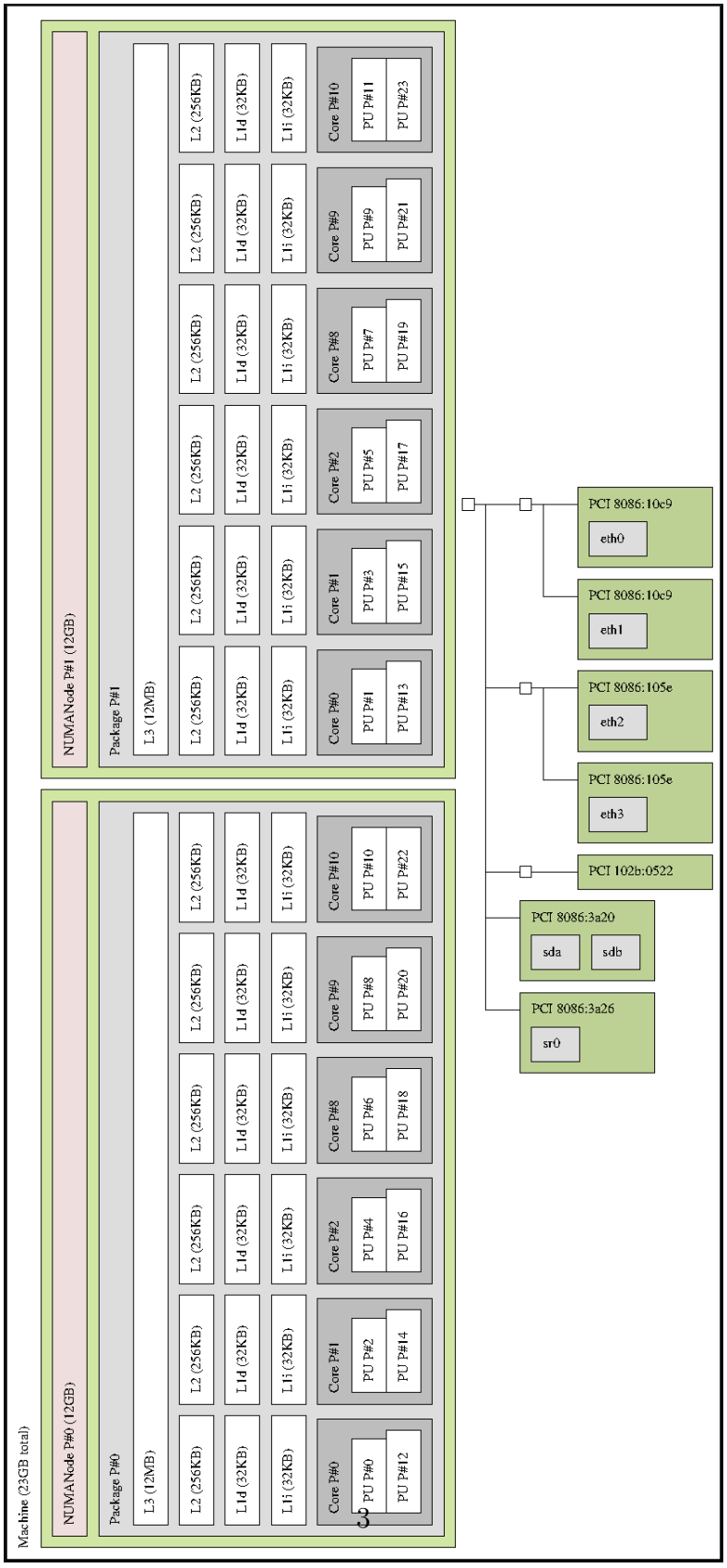


Figure 1: Graphical representation of the computer's architecture, using `lstopo`.

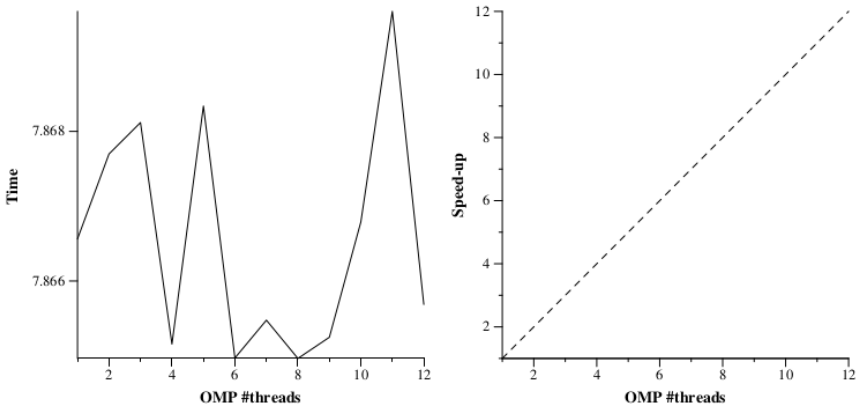
2 Timing sequential and parallel executions

2. Describe what do you need to add to your program to measure the elapsed execution time between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

To measure the elapsed time between two points of a given program, you need to include the `<sys/time.h>` library, which has an specific data structure for handling.

```
#include <sys/time.h>

double getusec_() {
    struct timeval time;
    gettimeofday(&time, NULL);
    return ((double)time.tv_sec * (double)1e6 + (double)time.tv_usec);
}

#define START_COUNT_TIME stamp = getusec_();

#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
                            stamp = stamp/1e6;\
                            printf ("%s%0.6f\n",(_m), stamp);

int main() {
    double stamp;       // Data structure that stores the elapsed time
    START_COUNT_TIME;

      /* Rest of the program goes here... */

    STOP_COUNT_TIME;
}
```

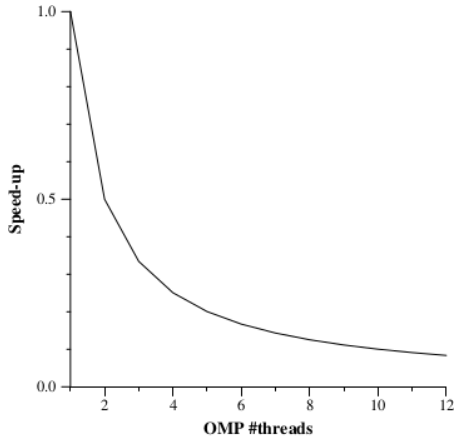
3. Plot the speed–up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for `pi_omp.c`. Reason about how the scalability of the program.



Min elapsed execution time

Speed-up wrt sequential time

On the strong scaling plot, we see an interesting plot on the minimum elapsed execution time. Generally, there is a huge spike in the execution time if the number of threads is odd, unless $\#_{threads} = 7$. In the case of even threads, the execution time is at its lowest when $\#_{threads} = \{6, 8\}$.



Speed-up wrt one thread, weak scaling

On the weak scaling plot, the speed-up decreases logarithmically each time we use another thread. This means that each time a new processor is added, the reductions in time won't increase as much as they did with the previous one.

3 Visualizing the task graph and data dependences

4. Include the source code for function `dot_product` in which you show the *Tareador* instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

```
void dot_product (long N, double A[N], double B[N], double *acc){
    double prod;
    int i;
    *acc=0.0;
    tareador_disable_object(acc);
    for (i=0; i<N; i++) {
        // tareador_start_task("dot_product");
        prod = my_func(A[i], B[i]);
        *acc += prod;
        //tareador_end_task("dot_product");
    }
    tareador_enable_object(acc);
}
```

5. Capture the task dependence graph for that task decomposition and the execution timelines (for 8 processors) that allow you to understand the potential parallelism attainable. Briefly comment the relevant information that is reported by the tools.

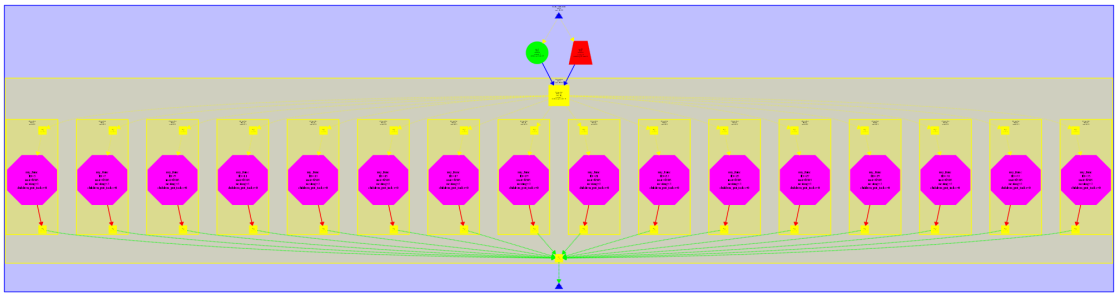


Figure 2: Ideal task dependency graph of `dot_product.c`

Figure 3: Execution timeline of `dot_product.c` for 8 threads

At first, two threads handle the functions `init_A(...)` and `init_B(...)`, respectively during the first 261 μ s (the other threads remain idle). Then each thread begins executing `dot_product`, which runs shortly before giving in to the more resource-consuming `my_func(...)`. This process is repeated twice; finally, `dot_product(...)` will be called again to synchronize the individual results into the thread 1.1, which then proceeds to return the parallelized result of the execution of `dot_product.c`.

4 Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for `3dfft_seq.c`, briefly commenting the evolution of the metrics with the different versions.

Version	T_1	T_∞	**Parallelism**
Seq.	593.772	593.772	1
V1	593.772	593.758	1
V2	593.772	315.523	1.8819
V3	593.772	109.063	5.4443
V4	593.772	60.148	9.8718

(Note that times are represented in *microseconds*).

Comparing all versions:

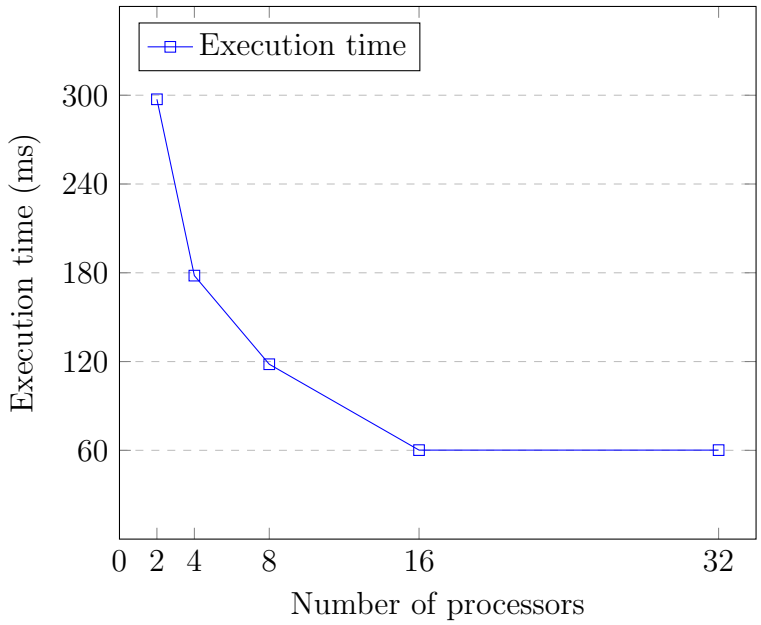
- We find that V1 produces a speedup of *just 14 instructions*, which produces no impact on the speedup compared to the sequential code.
- V2, instead, will execute using about 53.14% of seq's instructions. Unlike the previous versions, this one is parallelizable, and is almost 20,000 instructions away from being completely parallel for 2 cores.
- V3 executes about 18.37% of seq's instructions. It also improves by using almost three times less instructions than V2 when run with $\lceil 5.4413 \rceil = 6$ cores.
- V4 executes about 10.13% of seq's instructions. This is a huge improvement compared to seq's execution, yet it requires $\lceil 9.8718 \rceil = 10$ processors to take advantage of its potential. Moreover, the improvement from $V3 \rightarrow V4$ is not as remarkable as the one from $V2 \rightarrow V3$.

For all versions, using more cores than its parallelism value won't reduce the number of instructions of the program; in fact, they will remain the same. Also, each version improves on the previous one by having more parallelizable regions on its code.

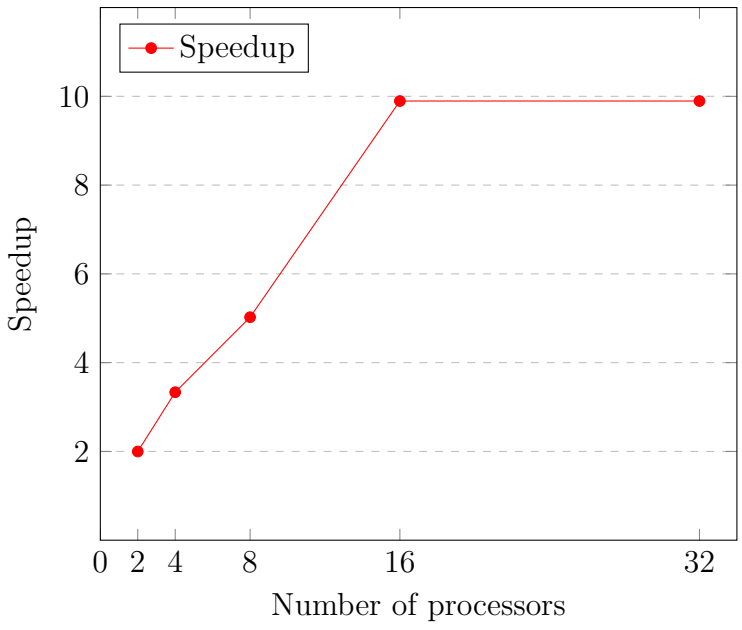
7. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version `v4` with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in

`3dfft_seq.c`, **using just 1 processor). Briefly comment the scalability behaviour shown on these two plots.**

Execution time plot by number of processors



Speedup plot in comparison to the sequential code



- Regarding execution time, we can find a logarithmic decrease on the execu-

tion time as more processors are executing the code. However, for 16 and 32 processors, the execution time remains the same: this is due to the potential parallelism of the program, $\lceil 9.8718 \rceil = 10$ processors. Once we have 10 or more processors, the program will execute as fast as it can possibly go.

- Regarding speedup, here speedup increases in a quasi-linear rate. Just like in the previous graph, there are no differences in speedup for 16 and 32 processors, and for the exact same reason: the potential parallelism.

5 Tracing sequential and parallel executions

8. From the instrumented version of `pi_seq.c`, and using the appropriate `Paraver` configuration file, obtain the value of the parallel fraction ϕ for this program when executed with 100,000,000 iterations, showing the steps you followed to obtain it. Clearly indicate which `Paraver` configuration file(s) did you use.

Figure 4: Execution timeline of `pi_seq.c` for 8 threads and 100,000,000 iterations

$$\phi = \frac{T_{par}}{T_1} = \frac{T_{par}}{T_{seq} + T_{par}} = \frac{0ms}{1067.048556ms + 0ms} = 0$$

Due to the fact that this code can't be parallelizable, $T_{par} = 0$.

- **Paraver configuration file used:** `thread_taskExecution.cfgs`

9. From the instrumented version of `pi_omp.c`, and using the appropriate `Paraver` configuration file, show a profile of the % of time spent in the different OpenMP states when using 8 threads and for 100,000,000 iterations. Clearly indicate which `Paraver` configuration file(s) did you use and your own conclusions from that profile.

Figure 5: Execution timeline of `pi_omp.c` for 8 threads and 100,000,000 iterations

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	368,835,871 ns	-	86,276,039 ns	3,656,681 ns	35,093 ns	2,298 ns
THREAD 1.1.2	151,471,773 ns	250,487,467 ns	36,713,233 ns	-	9,598 ns	-
THREAD 1.1.3	123,010,249 ns	250,405,286 ns	65,162,969 ns	-	6,812 ns	-
THREAD 1.1.4	138,039,304 ns	250,405,279 ns	50,231,224 ns	-	7,752 ns	-
THREAD 1.1.5	138,402,500 ns	250,405,244 ns	49,861,334 ns	-	6,283 ns	-
THREAD 1.1.6	188,138,318 ns	250,405,296 ns	34,469 ns	-	6,025 ns	-
THREAD 1.1.7	126,606,611 ns	250,480,082 ns	61,579,005 ns	-	5,548 ns	-
THREAD 1.1.8	122,758,088 ns	250,404,859 ns	65,499,531 ns	-	5,375 ns	-
Total	1,357,262,714 ns	1,752,993,513 ns	415,357,804 ns	3,656,681 ns	82,486 ns	2,298 ns
Average	169,657,839.25 ns	250,427,644.71 ns	51,919,725.50 ns	3,656,681 ns	10,310.75 ns	2,298 ns
Maximum	368,835,871 ns	250,487,467 ns	86,276,039 ns	3,656,681 ns	35,093 ns	2,298 ns
Minimum	122,758,088 ns	250,404,859 ns	34,469 ns	3,656,681 ns	5,375 ns	2,298 ns
StDev	77,901,997.75 ns	35,554.70 ns	23,884,769.84 ns	0 ns	9,454.44 ns	0 ns
Avg/Max	0.46	1.00	0.60	1	0.29	1

Figure 6: Execution profile of `pi_omp.c` for 8 threads and 100,000,000 iterations

$$\phi = \frac{T_{par}}{T_1} = \frac{T_{par}}{T_{seq} + T_{par}} = \frac{188.138318ms}{188.138318ms + 270.667664ms} = 0.41,$$

Where

- $T_{par} = max\{T_{ParallelExecution}\} = 188.138318ms$
- $T_{seq} = T_{FullProgram} - T_{par} = 270.667664ms$
- **Paraver configuration file used:** `thread_taskExecution.cfgs` and `OMP_state_profile.cfgs`, for each respective graph.

Regarding the profile, the first thread runs the longest, because it's the one which is also executing the (initial) sequential code. Then, it executes the scheduling and fork for the other 7 threads. Synchronization stands for the time between the end of a task in a given thread and the wait until the very last the threads ends its execution; it's inversely proportional to the running time of a parallel task (with thread #1 being the main exception). Once it ends, it synchronizes to thread #1 to save the obtained partial results into it, then execute another piece of sequential code until its completion.