PARALLELISM
1202

First Deliverable

Héctor Ramón Jiménez

Alvaro Espuña Buxó

March 13, 2014

Facultat d'Informàtica de Barcelona

Node architecture and memory

1. Describe (better if you do a simple drawing) 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).

   - **2 sockets**
   - **6 cores per socket**
   - **2 threads per core**
   - **3 cache levels: L1, L2 and L3**
     - **L1 (32 KBytes) and L2 (256 KBytes) private per core**
     - **L3 (12 MBytes) shared between cores of the same socket**
   - **Main memory (23.49 GBytes)**

Timing sequential and parallel executions

2. Indicate the library header where the structure struct timeval is declared and which are its fields.

   It is defined in `sys/time.h`. Its fields are:

   - **tv_sec: Time in seconds**
   - **tv_usec: Time in microseconds**

3. Plot the execution time when varying the number of threads for `pi_omp.c`. Reason about how the number of threads influence on the execution time.

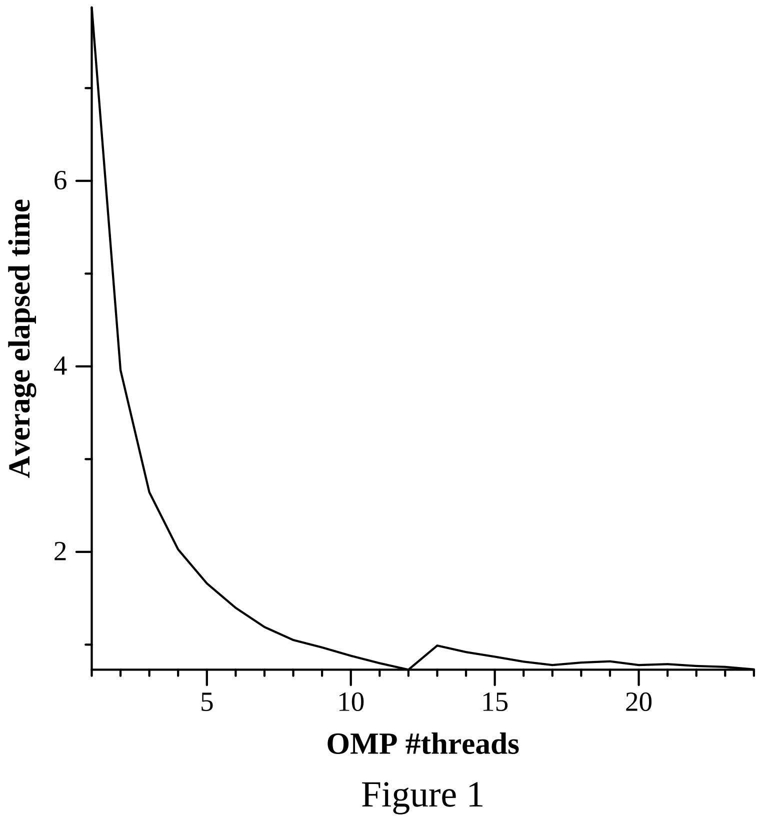

Figure 1: Figure showing execution time as function of the number of threads used

When using from 1 to 12 threads the execution time follows the formula $\frac{execution\,time\,with\,1\,thread}{number\,of\,threads}$, as expected.

When more than 12 threads are used there is a slight overhead. Again, this was expected because, despite that 2 threads per core are supported, the threads in a core share some resources (like the two first levels of caches).

Tracing parallel executions

4. From the two instrumented `OpenMP` versions (`pi_omp_sum_local.c` and `pi_omp_critical.c`), show a profile of the % of time spent in the different `OpenMP` states. Reason about the differences observed and guess which is the construct in the source code that is the cause of these differences.

	critical	sum_local
Running	9.79	85.12
Not created	0.01	5.40
Synchronization	87.15	5.94
Scheduling and Fork/Join	0.26	7.89
I/O	3.01	2.55

The differences are caused by the moment where the mutual exclusions are performed:

- **The "critical" version performs mutual exclusions in each iteration. Therefore, every thread needs to be synchronized with all the other threads before performing an iteration!**
- **The "sum_local" version only performs** *num_threads* **exclusions to reduce the local sums of each thread. Thus, this version can perform the most part of the computation without any synchronization with the other threads.**

5. For the two `OpenMP` versions provided, fill in the following table with the time elapsed in each part of the code and the total elapsed time of the instrumented code (using 8 threads).

	REST_MAIN	PI_COMPUTATION	TIMING	Total
critical	5540.00	101 118 642.50	24 749.00	132 382 580.88
sum_local	5268.00	190 077.88	16 061.00	3 767 231.88
SPEEDUP	1.05	531.98	1.54	35.14

It is clear that the PI_COMPUTATION has been performed faster in the sum_local version, avoiding the synchronization in each iteration and obtaining a huge speedup.

Visualizing the task graph and data dependences

6. Include the source code for function dot_product in which you show the instrumentation that has been added in order to identify tasks and filter the analysis of variable(s) that cause the dependence(s).

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <tareador_hooks.h>

#define size 16
double A[size]; // vector A globally declared
double * B;  // vector B dynamically allocated in main program

double my_func (double Ai, double Bi){
double comp;

   comp = Ai * Bi;
   int i;
   for (i=0; i < 500; i++)
        comp += i;

   return (comp);
}

void dot_product (long N, double A[N], double B[N], double *acc){
        double prod;

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

int main(int argc, char **argv) {
```

```
    struct timeval start;
    struct timeval stop;
    unsigned long elapsed;
    double result;

    double *B = malloc(size*sizeof(double));

    tareador_ON ();

    int i;

    tareador_start_task("init_A");
    for (i=0; i< size; i++) A[i]=i;
    tareador_end_task();

    tareador_start_task("init_B");
    for (i=0; i< size; i++) B[i]=2*i;
    tareador_end_task();

    gettimeofday(&start,NULL);

    dot_product (size, A, B, &result);

    tareador_OFF ();

    gettimeofday(&stop,NULL);
    elapsed = 1000000 * (stop.tv_sec - start.tv_sec);
    elapsed += stop.tv_usec - start.tv_usec;
    printf("Result of Dot product i= %le\n", result);
    printf("Execution time (us): %lu \n", elapsed);

    return 0;
}
```

7. Capture the task dependence graph and execution timeline (for 8 processors) for that task decomposition.

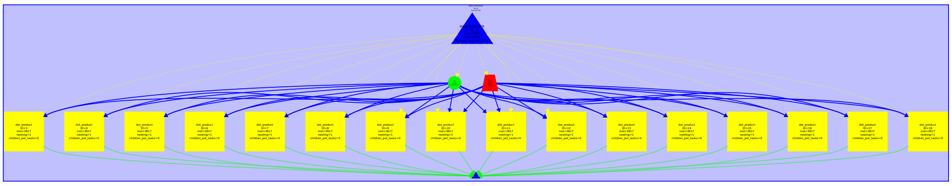

Figure 2: Figure showing the entire task dependency graph of `dot_product.c`

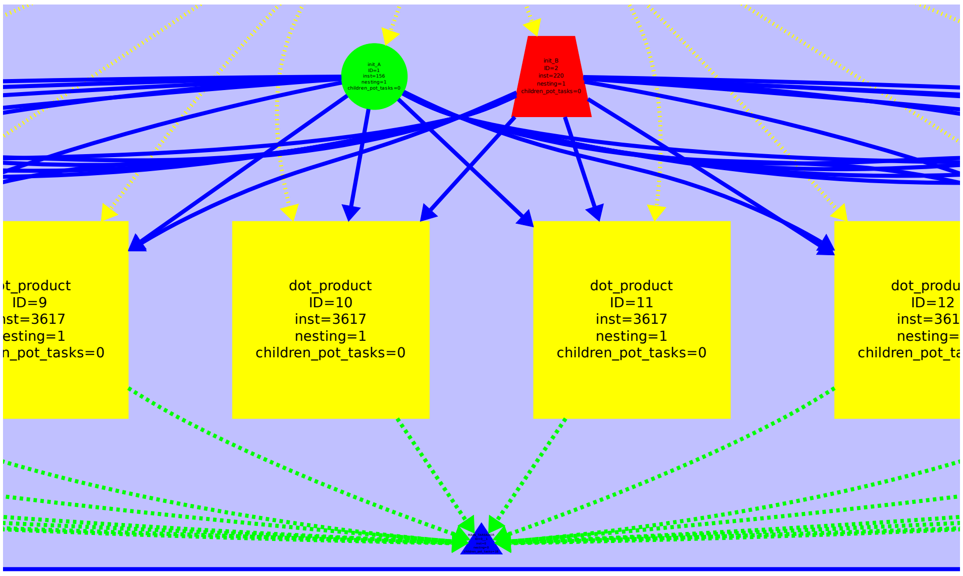

Figure 3: Figure zooming the center section of the task dependency graph of `dot_product.c`

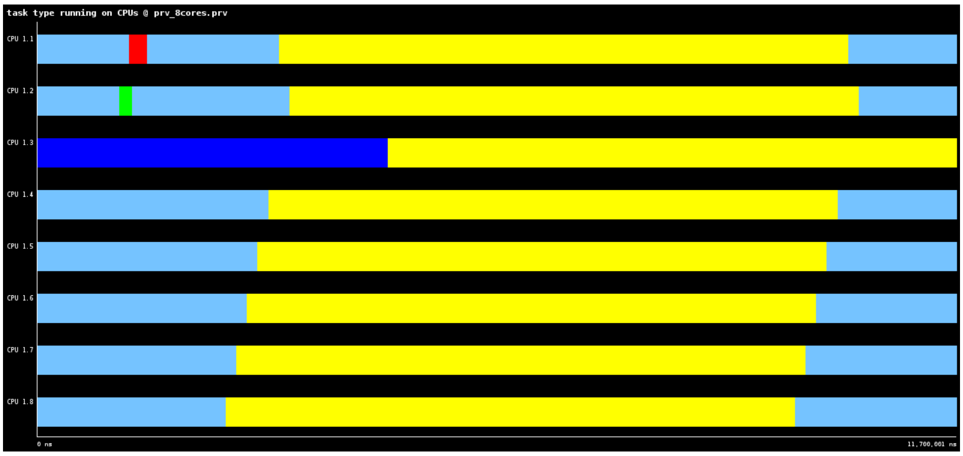

Figure 4: Figure showing the execution timeline of `dot_product.c` with 8 processors

Analysis of task decomposition

8. Complete the following table for the initial and different versions generated for `3dfft_seq.c`.

Version	T_1	T_∞	Parallelism
sqe	1073956	806555	1.331
v1	1075095	807694	1.331
v2	1079800	707042	1.527
v3	1086061	391033	2.777
v4	1087727	323330	3.364

9. 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).

Processors	Time (ms)	Speedup
1	1073.956	1
2	546.488	1.965
4	403.891	2.659
8	350.956	3.060
16	322.078	3.334
32	322.078	3.334

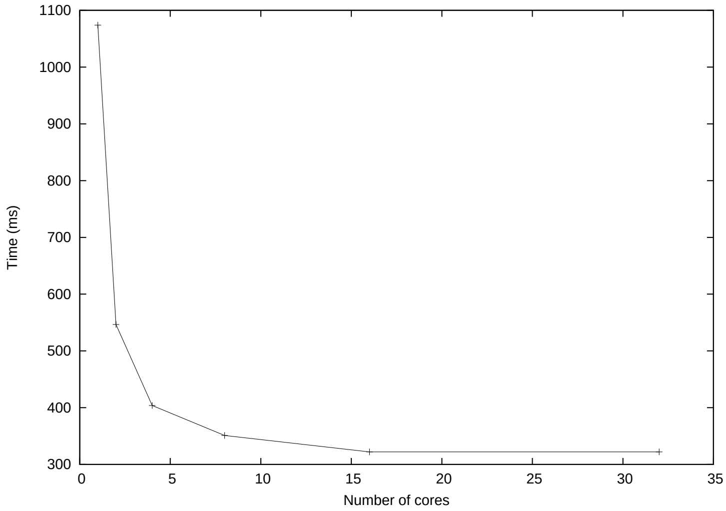

Figure 5: Figure showing the execution timeline as function of the number of cores used.

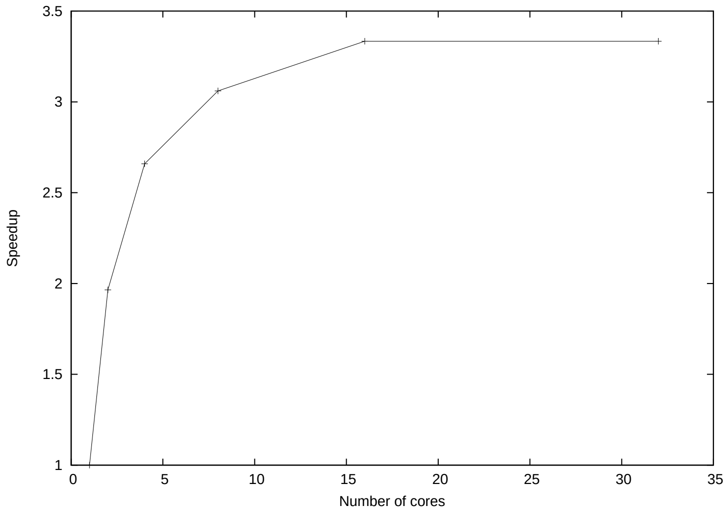

Figure 6: Figure showing the speedup as function of the number of cores used. We can see that the theoretical limit is almost reached.