PACO Laboratory

Assignment Lab 4:

Divide and Conquer parallelism with OpenMP: Sorting

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q2), Jesus Labarta, Josep Ramon Herrero (Q1), Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera, Jordi Tubella and Gladys Utrera

Fall 2023-24

Adrián García Campillo i Oriol Amate Sabat

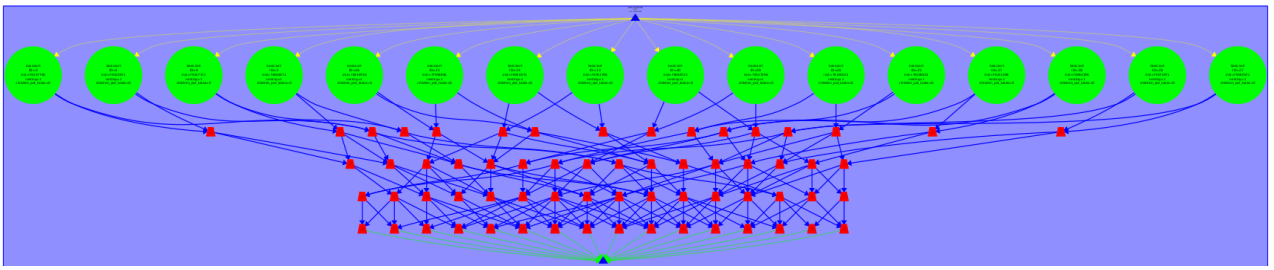
ÍNDEX

2.2 Task decomposition analysis with Tareador

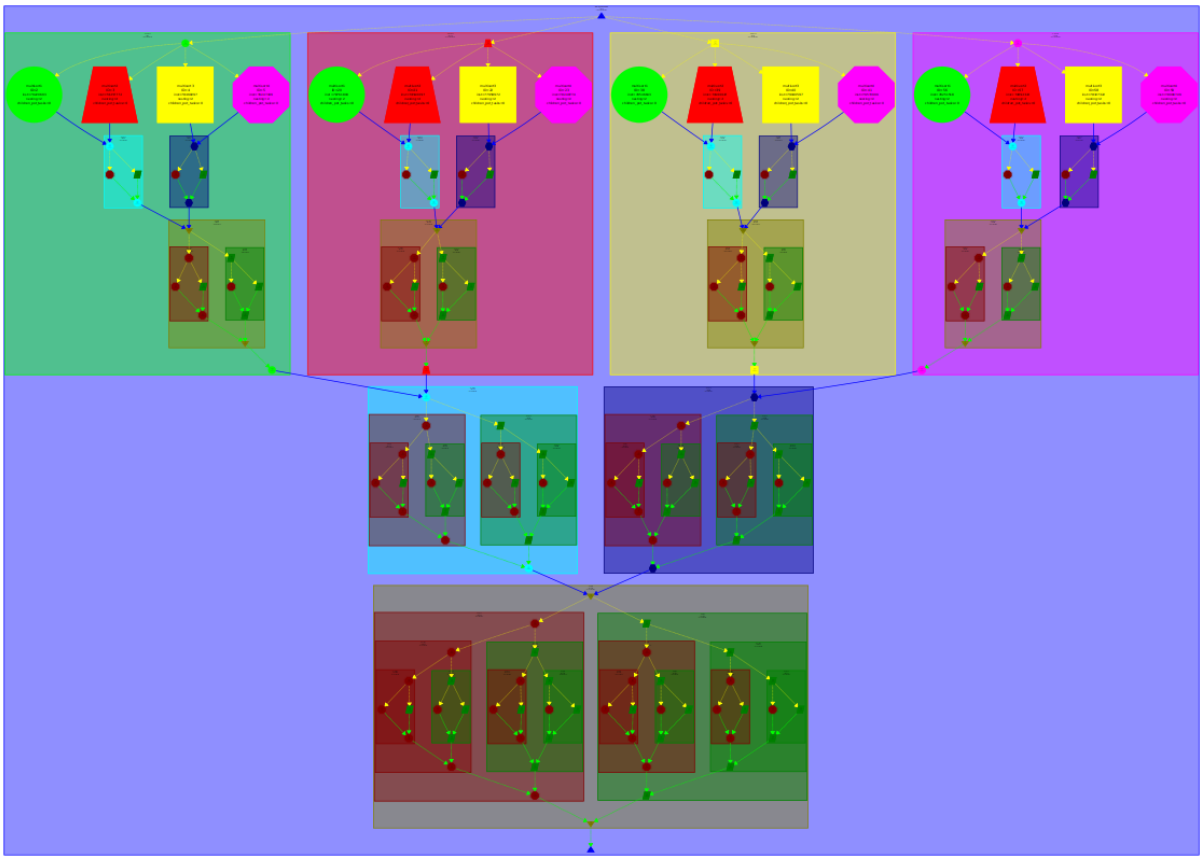
For the deliverable: Include both tareador codes (in the .zip file).

For the deliverable: Save the TDF graphs generated by Tareador to include it in the deliverable.

Leaf



Tree



For the deliverable: Identify the points of the code where you should include synchronizations to guarantee the dependences for each task decomposition strategy.

Which directives/annotations/clauses will you use to guarantee those dependences in the OpenMP implementations?

En ambas tácticas, la dependencia surge en la función merge debido a su rol en la organización de las dos secciones del vector original. Contrariamente, la función multisort no genera tales dependencias al ser independiente de estas secciones del vector. Además, se observa que multisort requiere más tiempo debido al incremento en el número de tareas generadas.

3 Shared-memory parallelisation with

OpenMP tasks

Leaf

For the deliverable: Include the three tables generated. Which of the factors do you think is making the parallelisation efficiency so low? Several options to think about: parallel fraction, in-execution efficiency (related with the overheads of sync and sched reported in the third table), number of tasks and their execution time, load balancing, ...

```
paco1106@boada-7:~/lab4$ cat multisort-omp_2_boada-11.times.txt
********************************************************************************
Problem size (in number of elements): N=1024, MIN_SORT_SIZE=256, MIN_MERGE_SIZE=256
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************************
Initialization time in seconds: 0.020411
Multisort execution time: 0.114807
Check sorted data execution time: 0.000777
Multisort program finished
********************************************************************************
paco1106@boada-7:~/lab4$
```

```
paco1106@boada-7:~/lab4$ cat speedup-global-multisort-omp-1024-256-50-boada-12.txt
1    .97708621060061995519
2    1.02827428054649397629
3    1.10870276858784607348
4    1.04916195962378808750
5    1.05817285001096848388
6    .85084479413321075221
7    .79686623948738486183
8    .77045046440377718298
9    .75043371676409579483
10    .75834536247004625858
11    .73479693861536046973
12    .73061158921399885255
13    .74282407731470463337
14    .73096386095421775267
15    .74565648143377100323
16    .73277755016549203382
17    .75354566283539414137
18    .75774972390418523173
19    .73653428834506276518
20    .73539099783330792436
paco1106@boada-7:~/lab4$
```

```
paco1106@boada-7:~/lab4$ cat speedup-partial-multisort-omp-1024-256-50-boada-12.txt
1    .96907250083488771778
2    1.04227844552685986613
3    1.15337411417990751601
4    1.07161579132251875737
5    1.07890417418588412394
6    .80984827342096127534
7    .75252371916508538899
8    .71267948172685831351
9    .69190239136503210198
10    .70191564552003256656
11    .67434109845264410814
12    .67386366854521254000
13    .68619975660539505482
14    .67711588041455377418
15    .68847905466792433176
16    .67483451596984702298
17    .69723039416777036767
18    .70127435840003772383
19    .68092924228636184131
20    .67488045198509266034
paco1106@boada-7:~/lab4$
```

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.20	0.20	0.21	0.21	0.23
Speedup	1.00	1.01	0.97	0.96	0.90
Efficiency	1.00	0.25	0.12	0.08	0.06

Table 1: Analysis done on Mon Nov 27 02:18:56 PM CET 2023, paco1106

Overview of the Efficiency metrics in parallel fraction, ϕ=89.53%					
Number of processors	1	4	8	12	16
Global efficiency	91.69%	23.18%	11.14%	7.28%	5.08%
Parallelization strategy efficiency	91.69%	39.29%	21.08%	14.79%	11.19%
Load balancing	100.00%	92.36%	46.23%	29.67%	21.96%
In execution efficiency	91.69%	42.54%	45.60%	49.84%	50.94%
Scalability for computation tasks	100.00%	59.01%	52.86%	49.26%	45.37%
IPC scalability	100.00%	52.09%	49.25%	48.77%	47.04%
Instruction scalability	100.00%	113.49%	112.59%	111.85%	112.03%
Frequency scalability	100.00%	99.80%	95.32%	90.30%	86.09%

Table 2: Analysis done on Mon Nov 27 02:18:56 PM CET 2023, paco1106

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.76	0.76	0.81	0.79
LB (time executing explicit tasks)	1.0	0.74	0.81	0.86	0.86
Time per explicit task (average us)	2.58	4.14	4.28	4.41	4.47
Overhead per explicit task (synch %)	0.96	155.26	479.76	792.98	1180.07
Overhead per explicit task (sched %)	9.97	41.01	32.11	28.1	29.37
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Mon Nov 27 02:18:56 PM CET 2023, paco1106

Podem veure que tenim una baixada d'eficàcia. Això es pot deure a sobrecàrrega per sincronització com indica la taula. També pot ser perquè el treball no és distribueix correctament.

For the deliverable: Include parts (zoom in) of the Paraver visualisations that help you explain the lack of scalability. In particular, we think it would be good to show those parts that show examples of: 1) the amount of task executed in parallel, 2) which threads are executing tasks and 3) which thread/s is/are creating tasks?. Then, you may comment/answer the following questions: Is the program generating enough tasks to simultaneously feed all threads? How many? How many threads are creating tasks?

Podem veure que només el thread 1 crea tasques, les quals si que es poden executar simultàniament a la resta de threads.

Tree

For the deliverable: Include the three tables generated with modelfactors. Which of the following options do you think is making the parallelisation better than the leaf version? Several options to think about: parallel fraction, in-execution efficiency (related with the overheads of sync and sched reported in the third table), number of tasks and their execution time, load balancing, ...

```
paco1106@boada-7:~/lab4$ cat multisort-omp_2_boada-11.times.txt
********************************************************************************
Problem size (in number of elements): N=1024, MIN_SORT_SIZE=256, MIN_MERGE_SIZE=256
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************************
Initialization time in seconds: 0.020410
Multisort execution time: 0.089054
Check sorted data execution time: 0.000665
Multisort program finished
********************************************************************************
paco1106@boada-7:~/lab4$
```

```
paco1106@boada-7:~/lab4$ cat speedup-global-multisort-omp-1024-256-50-boada-11.txt
1    .96490679306797880211
2    1.23350737832870665856
3    1.54435437285785805060
4    1.81659532060428982356
5    2.00238425145738930706
6    2.14047050346272400955
7    2.29506531412301661110
8    2.38753609382907153275
9    2.43375815505993020785
10    2.50760524629117228657
11    2.60062255800003242489
12    2.61090855822129626615
13    2.69111010451792574697
14    2.70961148648648648648
15    2.71792134736271370236
16    2.74470851941207672432
17    2.79750610394140216253
18    2.80572657944448331351
19    2.84256880083641969839
20    2.83538374518329974900
paco1106@boada-7:~/lab4$
```

```
paco1106@boada-7:~/lab4$ cat speedup-partial-multisort-omp-1024-256-50-boada-11.txt
1    .94912201353678180072
2    1.33980375904478146911
3    1.93495422045358956938
4    2.55702345069368733833
5    3.00638399477216176137
6    3.50941790869616242225
7    3.97035881435257410296
8    4.37445143358689291983
9    4.78230449384295538141
10    5.15115628095258602127
11    5.50409534327259341063
12    5.83373975809598127194
13    6.22508456934686442883
14    6.41780233930679257431
15    6.77129917916784602320
16    6.78628162941109724270
17    7.22312801932367149758
18    7.37590183141148177838
19    7.69079920272616215521
20    7.82001830543933054393
paco1106@boada-7:~/lab4$
```

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.26	0.24	0.23	0.23	0.23
Speedup	1.00	1.09	1.11	1.11	1.10
Efficiency	1.00	0.27	0.14	0.09	0.07

Table 1: Analysis done on Mon Nov 27 02:34:28 PM CET 2023, paco1106

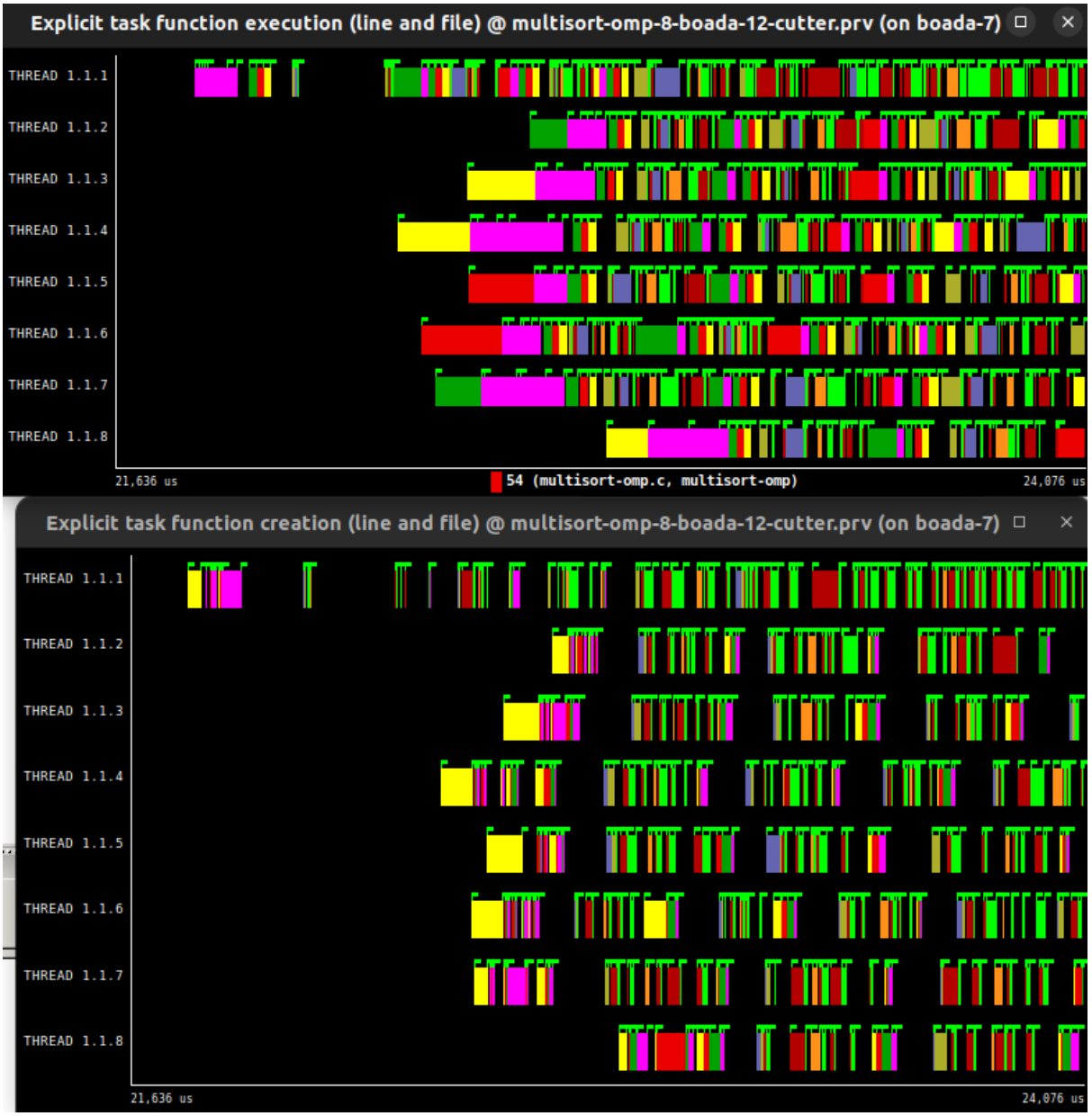
Overview of the Efficiency metrics in parallel fraction, ϕ=91.62%					
Number of processors	1	4	8	12	16
Global efficiency	88.11%	24.09%	12.30%	8.20%	6.09%
Parallelization strategy efficiency	88.11%	34.66%	20.04%	13.78%	10.57%
Load balancing	100.00%	95.16%	92.42%	90.34%	89.40%
In execution efficiency	88.11%	36.43%	21.68%	15.26%	11.82%
Scalability for computation tasks	100.00%	69.50%	61.39%	59.49%	57.65%
IPC scalability	100.00%	58.58%	54.73%	56.36%	56.68%
Instruction scalability	100.00%	121.36%	121.27%	121.29%	121.22%
Frequency scalability	100.00%	97.77%	92.50%	87.03%	83.91%

Table 2: Analysis done on Mon Nov 27 02:34:28 PM CET 2023, paco1106

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.96	0.97	0.97	0.98
LB (time executing explicit tasks)	1.0	0.99	0.99	0.99	0.99
Time per explicit task (average us)	1.76	5.32	9.37	13.53	17.69
Overhead per explicit task (synch %)	1.08	58.95	75.98	82.45	87.13
Overhead per explicit task (sched %)	14.78	46.44	66.98	77.51	83.4
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Mon Nov 27 02:34:28 PM CET 2023, paco1106

For the deliverable: Include sections of the Paraver visualisations that help you explain the lack of scalability. In particular, we think it would be good to show those parts that show examples of: 1) the amount of task executed in parallel, 2) which threads are executing tasks and 3) which thread/s is/are creating tasks?. Then, you may comment/answer the following questions: Is the program generating enough tasks to simultaneously feed all threads? How many? How many threads are creating tasks?



Podem veure com tots el threads generen tasques, i de la mateixa manera tots execute tasques simultàniament.

For the deliverable: On the other hand, the tree strategy is not showing a good parallel performance neither. Is the granurality of both (leaf and tree) strategies influencing the parallel performance (see overheads in modelfactor tables)? What is the number of tasks created (see modelfactor tables)?

2730

3.3 Task granularity control: the cut–off mechanism

For the deliverable: Include the postscript generated. Which is the best value for cut-off for this problem size and 8 threads?

```
paco1106@boada-7:~/lab4$ cat multisort-omp_1_boada-11.times.txt
********************************************************************************
Problem size (in number of elements): N=1024, MIN_SORT_SIZE=256, MIN_MERGE_SIZE=256
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=1
********************************************************************************
Initialization time in seconds: 0.020052
Multisort execution time: 0.133950
Check sorted data execution time: 0.000410
Multisort program finished
********************************************************************************
paco1106@boada-7:~/lab4$ sbatch ./submit-omp.sh multisort-omp 8
Submitted batch job 146196
paco1106@boada-7:~/lab4$ cat multisort-omp_8_boada-11.times.txt
********************************************************************************
Problem size (in number of elements): N=1024, MIN_SORT_SIZE=256, MIN_MERGE_SIZE=256
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=8
********************************************************************************
Initialization time in seconds: 0.020693
Multisort execution time: 0.132703
Check sorted data execution time: 0.000409
Multisort program finished
********************************************************************************
paco1106@boada-7:~/lab4$
```

```
paco1106@boada-7:~/lab4$ cat elapsed.txt
0
0.050968
1
0.021745
2
0.014756
3
0.011506
4
0.010940
5
0.010820
6
0.011863
7
0.013206
8
0.013145
9
0.014416
10
0.014936
11
0.016292
12
0.016217
13
0.015927
14
0.015959
15
0.015634
paco1106@boada-7:~/lab4$
```

El millor de cutoff es 5.

For the deliverable: Include the tables of modelfactors for cut–off level equal to 4 and the information of the number of tasks generated for each of the cut–off levels used.

Does it significantly change with the number of threads used? Which is the best value for cut-off?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.32	0.25	0.23	0.23	0.23
Speedup	1.00	1.26	1.39	1.38	1.41
Efficiency	1.00	0.32	0.17	0.12	0.09

Table 1: Analysis done on Mon Nov 27 02:02:18 PM CET 2023, paco1106

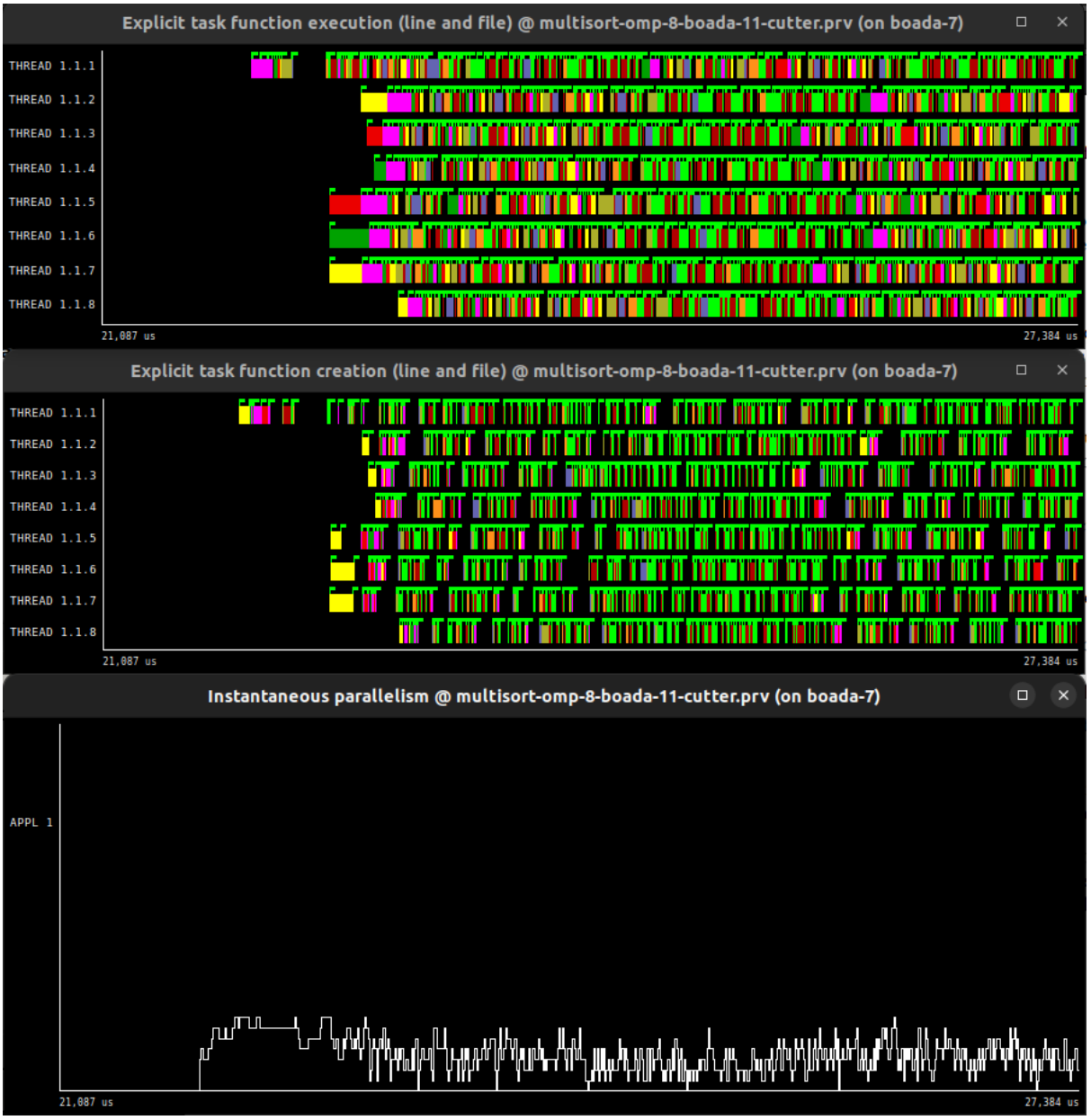
Overview of the Efficiency metrics in parallel fraction, ϕ=93.62%					
Number of processors	1	4	8	12	16
Global efficiency	75.08%	24.25%	13.52%	8.93%	6.84%
Parallelization strategy efficiency	75.08%	38.89%	23.90%	16.59%	12.88%
Load balancing	100.00%	96.08%	93.78%	92.03%	89.02%
In execution efficiency	75.08%	40.48%	25.48%	18.03%	14.47%
Scalability for computation tasks	100.00%	62.34%	56.58%	53.79%	53.10%
IPC scalability	100.00%	53.47%	51.05%	51.80%	52.75%
Instruction scalability	100.00%	118.92%	118.86%	118.82%	118.87%
Frequency scalability	100.00%	98.04%	93.24%	87.39%	84.68%

Table 2: Analysis done on Mon Nov 27 02:02:18 PM CET 2023, paco1106

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.96	0.98	0.98	0.97
LB (time executing explicit tasks)	1.0	0.99	0.98	0.96	0.97
Time per explicit task (average us)	1.79	6.51	11.74	17.84	23.06
Overhead per explicit task (synch %)	10.72	51.05	58.6	61.88	64.07
Overhead per explicit task (sched %)	13.36	36.2	49.07	55.76	59.39
Number of taskwait/taskgroup (total)	49152.0	49152.0	49152.0	49152.0	49152.0

Table 3: Analysis done on Mon Nov 27 02:02:18 PM CET 2023, paco1106

For the deliverable: Include the Paraver trace you have analyzed. Is the instantaneous paral- lelism achieved larger than one along all the execution trace?



Si

4 Shared-memory parallelisation with OpenMP task using dependencies

For the deliverable: Include the tables of modelfactors, scalability plots and Paraver windows to support the comparison. Are they better or worse compare to OpenMP versions of previous chapter in terms of performance? In terms of programmability, was this new version simpler to code?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.32	0.25	0.23	0.23	0.23
Speedup	1.00	1.27	1.39	1.41	1.37
Efficiency	1.00	0.32	0.17	0.12	0.09

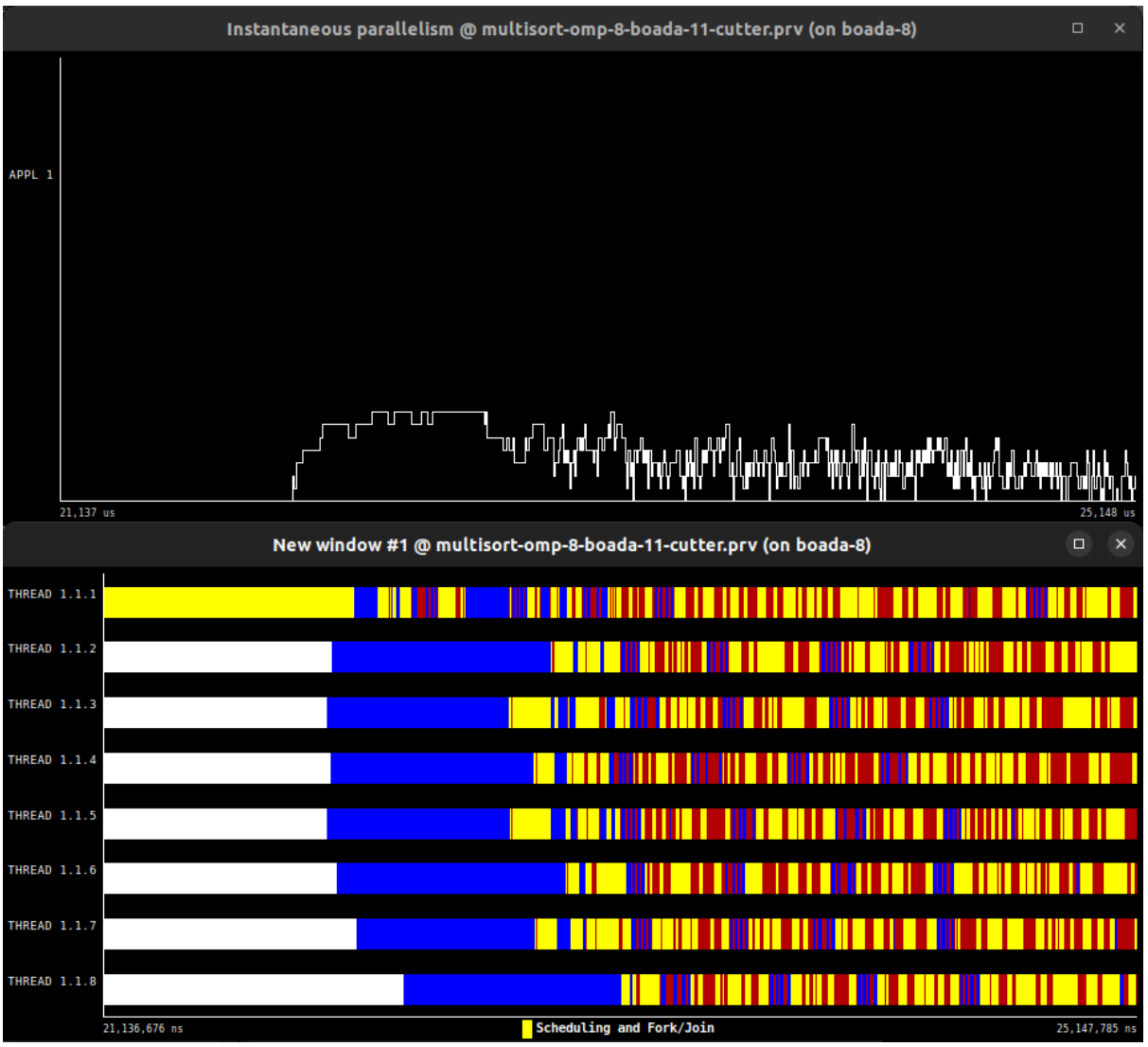
Table 1: Analysis done on Tue Nov 28 09:36:21 PM CET 2023, paco1106

Overview of the Efficiency metrics in parallel fraction, ϕ=93.25%					
Number of processors	1	4	8	12	16
Global efficiency	74.47%	24.02%	13.36%	8.97%	6.56%
Parallelization strategy efficiency	74.47%	37.85%	23.91%	16.42%	12.39%
Load balancing	100.00%	94.86%	93.75%	91.04%	89.45%
In execution efficiency	74.47%	39.90%	25.50%	18.03%	13.85%
Scalability for computation tasks	100.00%	63.46%	55.87%	54.63%	52.96%
IPC scalability	100.00%	53.95%	50.79%	52.51%	52.63%
Instruction scalability	100.00%	118.47%	118.58%	118.51%	118.50%
Frequency scalability	100.00%	99.30%	92.78%	87.79%	84.92%

Table 2: Analysis done on Tue Nov 28 09:36:21 PM CET 2023, paco1106

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.97	0.97	0.97	0.96
LB (time executing explicit tasks)	1.0	0.99	0.99	0.97	0.98
Time per explicit task (average us)	1.75	6.45	11.63	17.39	23.59
Overhead per explicit task (synch %)	11.57	51.94	59.39	62.93	65.25
Overhead per explicit task (sched %)	14.07	37.1	49.03	55.53	59.64
Number of taskwait/taskgroup (total)	49152.0	49152.0	49152.0	49152.0	49152.0

Table 3: Analysis done on Tue Nov 28 09:36:21 PM CET 2023, paco1106



Aquesta versió és millor que les anterior en tema d'eficiència. En termes de programació, la primera versió és més simple de programar.

Codis

2.2 Leaf

C/C++

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <tareador.h>

// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long
length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
	if (length < MIN_MERGE_SIZE*2L) {
	// Base case
	tareador_start_task("basicmerge");
	basicmerge(n, left, right, result, start, length);
	tareador_end_task("basicmerge");
	} else {
	// Recursive decomposition
	merge(n, left, right, result, start, length/2);
	merge(n, left, right, result, start + length/2, length/2);
	}
}
```

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0]);
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
        // Base case
        tareador_start_task("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
        }
}

static void initialize(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        if (i==0) {
        data[i] = rand();
        } else {
        data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
        }
}

static void clear(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        data[i] = 0;
        }
}
```

```
void check_sorted(long n, T data[n])
{
	int unsorted=0;
	for (int i=1; i<n; i++)
	if (data[i-1] > data[i]) unsorted++;
	if (unsorted > 0)
	printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
	else {
	//	printf ("data IS ordered; ");
	}
}

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

	/* Defaults for command line arguments */
	/* Importsant: all of them should be powers of two */
	N = 32 * 1024;
	MIN_SORT_SIZE = 1024;
	MIN_MERGE_SIZE = 1024;

	/* Process command-line arguments */
	for (int i=1; i<argc; i++) {
	if (strcmp(argv[i], "-n")==0) {
	N = atol(argv[++i]) * 1024;
	}
	else if (strcmp(argv[i], "-s")==0) {
	MIN_SORT_SIZE = atol(argv[++i]);
	}
	else if (strcmp(argv[i], "-m")==0) {
	MIN_MERGE_SIZE = atol(argv[++i]);
	}
	else {
	fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
```

```
	fprintf(stderr, "    -n to specify the size of the vector (in
Kelements) to sort (default 32)\n");
	fprintf(stderr, "    -s to specify the size of the vector (in elements)
that breaks recursion in the sort phase (default 1024)\n");
	fprintf(stderr, "    -m to specify the size of the vector (in elements)
that breaks recursion in the merge phase (default 1024)\n");
	return EXIT_FAILURE;
	}
	}

	fprintf(stdout,
"************************************************************************
**************\n");
	fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld,	MIN_MERGE_SIZE=%ld\n",	N/1024,	MIN_SORT_SIZE,
MIN_MERGE_SIZE);
	fprintf(stdout,
"************************************************************************
**************\n");

	T *data = malloc(N*sizeof(T));
	T *tmp = malloc(N*sizeof(T));

	initialize(N, data);
	clear(N, tmp);

	tareador_ON();
	multisort(N, data, tmp);
	tareador_OFF();

	check_sorted (N, data);

	fprintf(stdout, "Multisort program finished\n");
	fprintf(stdout,
"************************************************************************
**************\n");
	return 0;
```

```
}
```

2.2 Tree

```
C/C++
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <tareador.h>

// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long
length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
	if (length < MIN_MERGE_SIZE*2L) {
	// Base case
	basicmerge(n, left, right, result, start, length);
	} else {
	// Recursive decomposition
	tareador_start_task("merge1");
	merge(n, left, right, result, start, length/2);
	tareador_end_task("merge1");
	tareador_start_task("merge2");
```

```
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge2");
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("multisort1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("multisort1");
        tareador_start_task("multisort2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("multisort2");
        tareador_start_task("multisort3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("multisort3");
        tareador_start_task("multisort4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("multisort4");

  tareador_start_task("merge3");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge3");
  tareador_start_task("merge4");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("merge4");
  tareador_start_task("merge5");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge5");
        } else {
        // Base case
        basicsort(n, data);
        }
}

static void initialize(long length, T data[length]) {
```

```
        long i;
        for (i = 0; i < length; i++) {
        if (i==0) {
        data[i] = rand();
        } else {
        data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
        }
}

static void clear(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        data[i] = 0;
        }
}

void check_sorted(long n, T data[n])
{
        int unsorted=0;
        for (int i=1; i<n; i++)
        if (data[i-1] > data[i]) unsorted++;
        if (unsorted > 0)
        printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
        else {
        //     printf ("data IS ordered; ");
        }
}

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

        /* Defaults for command line arguments */
        /* Importsant: all of them should be powers of two */
        N = 32 * 1024;
        MIN_SORT_SIZE = 1024;
        MIN_MERGE_SIZE = 1024;
```

```
        /* Process command-line arguments */
        for (int i=1; i<argc; i++) {
        if (strcmp(argv[i], "-n")==0) {
        N = atol(argv[++i]) * 1024;
        }
        else if (strcmp(argv[i], "-s")==0) {
        MIN_SORT_SIZE = atol(argv[++i]);
        }
        else if (strcmp(argv[i], "-m")==0) {
        MIN_MERGE_SIZE = atol(argv[++i]);
        }
        else {
        fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
        fprintf(stderr, "    -n to specify the size of the vector (in
Kelements) to sort (default 32)\n");
        fprintf(stderr, "    -s to specify the size of the vector (in elements)
that breaks recursion in the sort phase (default 1024)\n");
        fprintf(stderr, "    -m to specify the size of the vector (in elements)
that breaks recursion in the merge phase (default 1024)\n");
        return EXIT_FAILURE;
        }
        }

        fprintf(stdout,
"**************************************************************************
**************\n");
        fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld,      MIN_MERGE_SIZE=%ld\n",      N/1024,      MIN_SORT_SIZE,
MIN_MERGE_SIZE);
        fprintf(stdout,
"**************************************************************************
**************\n");

        T *data = malloc(N*sizeof(T));
        T *tmp = malloc(N*sizeof(T));
```

```
        initialize(N, data);
        clear(N, tmp);

        tareador_ON();
        multisort(N, data, tmp);
        tareador_OFF();

        check_sorted (N, data);

        fprintf(stdout, "Multisort program finished\n");
        fprintf(stdout,
"********************************************************************************
**************\n");
        return 0;
}
```

3 Leaf

```
C/C++
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "omp.h"
#include "fake.h"

#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);

// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;
int CUTOFF;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long
length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
       if (length < MIN_MERGE_SIZE*2L) {
       // Base case
       #pragma omp task
       basicmerge(n, left, right, result, start, length);
       } else {
       // Recursive decomposition
       merge(n, left, right, result, start, length/2);
       merge(n, left, right, result, start + length/2, length/2);
       }
}

void multisort(long n, T data[n], T tmp[n]) {
       if (n >= MIN_SORT_SIZE*4L) {
       // Recursive decomposition
       multisort(n/4L, &data[0], &tmp[0]);
       multisort(n/4L, &data[n/4L], &tmp[n/4L]);
       multisort(n/4L, &data[n/2L], &tmp[n/2L]);
       multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
```

```
  #pragma omp taskwait
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

  #pragma omp taskwait
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
        // Base case
        #pragma omp task
        basicsort(n, data);
        }
}

static void initialize(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        if (i==0) {
        data[i] = rand();
        } else {
        data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
        }
}

static void clear(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        data[i] = 0;
        }
}

void check_sorted(long n, T data[n])
{
        int unsorted=0;
        for (int i=1; i<n; i++)
        if (data[i-1] > data[i]) unsorted++;
```

```
        if (unsorted > 0)
        printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
}

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

#ifdef _OPENMP
        TRACE_START_MARK;
#endif



        /* Defaults for command line arguments */
        /* Important: all of them should be powers of two */
        N = 32768 * 1024;
        MIN_SORT_SIZE = 1024;
        MIN_MERGE_SIZE = 1024;
        CUTOFF = 50;

        /* Process command-line arguments */
        for (int i=1; i<argc; i++) {
        if (strcmp(argv[i], "-n")==0) {
        N = atol(argv[++i]) * 1024;
        }
        else if (strcmp(argv[i], "-s")==0) {
        MIN_SORT_SIZE = atol(argv[++i]);
        }
        else if (strcmp(argv[i], "-m")==0) {
        MIN_MERGE_SIZE = atol(argv[++i]);
        }
#ifdef _OPENMP
        else if (strcmp(argv[i], "-c")==0) {
        CUTOFF = atoi(argv[++i]);
        }
#endif
        else {
```

```
#ifdef _OPENMP
      fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE] -c CUTOFF\n", argv[0]);
#else
      fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
#endif
      fprintf(stderr, "   -n to specify the size of the vector (in
Kelements) to sort (default 32768)\n");
      fprintf(stderr, "   -s to specify the size of the vector (in elements)
that breaks recursion in the sort phase (default 1024)\n");
      fprintf(stderr, "   -m to specify the size of the vector (in elements)
that breaks recursion in the merge phase (default 1024)\n");
#ifdef _OPENMP
      fprintf(stderr, "   -c to specify the cut off recursion level to stop
task generation in OpenMP (default 16)\n");
#endif
      return EXIT_FAILURE;
      }
      }

      fprintf(stdout,
"************************************************************************
**************\n");
      fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld,     MIN_MERGE_SIZE=%ld\n",     N/1024,     MIN_SORT_SIZE,
MIN_MERGE_SIZE);
#ifdef _OPENMP
      fprintf(stdout, "Cut-off level:                    CUTOFF=%d\n",
CUTOFF);
      fprintf(stdout, "Number of threads in OpenMP:
OMP_NUM_THREADS=%d\n", omp_get_max_threads());
#endif
      fprintf(stdout,
"************************************************************************
**************\n");
```

```
        T *data = malloc(N*sizeof(T));
        T *tmp = malloc(N*sizeof(T));

        double stamp;
        START_COUNT_TIME;

        initialize(N, data);
        clear(N, tmp);

        STOP_COUNT_TIME("Initialization time in seconds");

        START_COUNT_TIME;

        #pragma omp parallel
        #pragma omp single
        multisort(N, data, tmp);

        STOP_COUNT_TIME("Multisort execution time");

        START_COUNT_TIME;

        check_sorted (N, data);

        STOP_COUNT_TIME("Check sorted data execution time");

        fprintf(stdout, "Multisort program finished\n");
        fprintf(stdout,
"************************************************************************
**************\n");

#ifdef _OPENMP
        TRACE_END_MARK;
#endif

        return 0;
}
```

3 Tree

C/C++

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "omp.h"
#include "fake.h"

#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);

// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;
int CUTOFF;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long
length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
       if (length < MIN_MERGE_SIZE*2L) {
```

```
        // Base case
        basicmerge(n, left, right, result, start, length);
        } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, right, result, start, length/2);
        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2);
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp taskgroup
        {
        #pragma omp task
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
  }
  #pragma omp taskgroup
  {
  #pragma omp task
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        #pragma omp task
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        }

  #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
        // Base case
```

```
        basicsort(n, data);
        }
}

static void initialize(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        if (i==0) {
        data[i] = rand();
        } else {
        data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
        }
}

static void clear(long length, T data[length]) {
        long i;
        for (i = 0; i < length; i++) {
        data[i] = 0;
        }
}

void check_sorted(long n, T data[n])
{
        int unsorted=0;
        for (int i=1; i<n; i++)
        if (data[i-1] > data[i]) unsorted++;
        if (unsorted > 0)
        printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
}

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

#ifdef _OPENMP
        TRACE_START_MARK;
#endif
```

```
        /* Defaults for command line arguments */
        /* Important: all of them should be powers of two */
        N = 32768 * 1024;
        MIN_SORT_SIZE = 1024;
        MIN_MERGE_SIZE = 1024;
        CUTOFF = 50;

        /* Process command-line arguments */
        for (int i=1; i<argc; i++) {
        if (strcmp(argv[i], "-n")==0) {
        N = atol(argv[++i]) * 1024;
        }
        else if (strcmp(argv[i], "-s")==0) {
        MIN_SORT_SIZE = atol(argv[++i]);
        }
        else if (strcmp(argv[i], "-m")==0) {
        MIN_MERGE_SIZE = atol(argv[++i]);
        }
#ifdef _OPENMP
        else if (strcmp(argv[i], "-c")==0) {
        CUTOFF = atoi(argv[++i]);
        }
#endif
        else {
#ifdef _OPENMP
        fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE] -c CUTOFF\n", argv[0]);
#else
        fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
#endif
        fprintf(stderr, "   -n to specify the size of the vector (in
Kelements) to sort (default 32768)\n");
```

```
        fprintf(stderr, "    -s to specify the size of the vector (in elements)
that breaks recursion in the sort phase (default 1024)\n");
        fprintf(stderr, "    -m to specify the size of the vector (in elements)
that breaks recursion in the merge phase (default 1024)\n");
#ifdef _OPENMP
        fprintf(stderr, "    -c to specify the cut off recursion level to stop
task generation in OpenMP (default 16)\n");
#endif
        return EXIT_FAILURE;
        }
        }

        fprintf(stdout,
"********************************************************************************
**************\n");
        fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld,      MIN_MERGE_SIZE=%ld\n",      N/1024,      MIN_SORT_SIZE,
MIN_MERGE_SIZE);
#ifdef _OPENMP
        fprintf(stdout, "Cut-off level:                      CUTOFF=%d\n",
CUTOFF);
        fprintf(stdout, "Number of threads in OpenMP:
OMP_NUM_THREADS=%d\n", omp_get_max_threads());
#endif
        fprintf(stdout,
"********************************************************************************
**************\n");

        T *data = malloc(N*sizeof(T));
        T *tmp = malloc(N*sizeof(T));

        double stamp;
        START_COUNT_TIME;

        initialize(N, data);
        clear(N, tmp);
```

```
        STOP_COUNT_TIME("Initialization time in seconds");

        START_COUNT_TIME;

        #pragma omp parallel
        #pragma omp single
        multisort(N, data, tmp);

        STOP_COUNT_TIME("Multisort execution time");

        START_COUNT_TIME;

        check_sorted (N, data);

        STOP_COUNT_TIME("Check sorted data execution time");

        fprintf(stdout, "Multisort program finished\n");
        fprintf(stdout,
"************************************************************************
**************\n");

#ifdef _OPENMP
        TRACE_END_MARK;
#endif

        return 0;
}
```

3.3 Cutoff

```
C/C++
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include "omp.h"
#include "fake.h"

#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);

// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;
int CUTOFF;
int d;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start,
long length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length, int d) {
       if (length < MIN_MERGE_SIZE*2L) {
       // Base case
       basicmerge(n, left, right, result, start, length);
       } else {
```

```
        // Recursive decomposition
        if (!omp_in_final())
        {
         #pragma omp task final (d>=CUTOFF)
         merge(n, left, right, result, start, length/2, d+1);
         #pragma omp task final (d>=CUTOFF)
         merge(n, left, right, result, start + length/2, length/2, d+1);
         #pragma omp taskwait
    }else{
         merge(n, left, right, result, start, length/2, d+1);
         merge(n, left, right, result, start + length/2, length/2, d+1);
    }

        }
}

void multisort(long n, T data[n], T tmp[n], int d) {
       if (n >= MIN_SORT_SIZE*4L) {
       // Recursive decomposition
       if (!omp_in_final()) {
         #pragma omp task final (d>=CUTOFF)
         multisort(n/4L, &data[0], &tmp[0], d+1);
         #pragma omp task final (d>=CUTOFF)
         multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);
         #pragma omp task final (d>=CUTOFF)
         multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);
         #pragma omp task final (d>=CUTOFF)
         multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], d+1);
         #pragma omp taskwait

         #pragma omp task final (d>=CUTOFF)
         merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);
         #pragma omp task final (d>=CUTOFF)
         merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, d+1);
         #pragma omp taskwait

         #pragma omp task final (d>=CUTOFF)
```

```
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);
        #pragma omp taskwait
        }
    else{
        multisort(n/4L, &data[0], &tmp[0], d+1);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], d+1);

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, d+1);

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);
        }
       } else {
       // Base case
       basicsort(n, data);
       }
}

static void initialize(long length, T data[length]) {
       long i;
       for (i = 0; i < length; i++) {
       if (i==0) {
              data[i] = rand();
       } else {
              data[i] = ((data[i-1]+1) * i * 104723L) % N;
       }
       }
}

static void clear(long length, T data[length]) {
       long i;
       for (i = 0; i < length; i++) {
       data[i] = 0;
       }
}
```

```
void check_sorted(long n, T data[n])
{
	int unsorted=0;
	for (int i=1; i<n; i++)
	if (data[i-1] > data[i]) unsorted++;
	if (unsorted > 0)
	printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
}

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

#ifdef _OPENMP
	TRACE_START_MARK;
#endif



	/* Defaults for command line arguments */
	/* Important: all of them should be powers of two */
	N = 32768 * 1024;
	MIN_SORT_SIZE = 1024;
	MIN_MERGE_SIZE = 1024;
	CUTOFF = 50;
	d = 0;

	/* Process command-line arguments */
	for (int i=1; i<argc; i++) {
	if (strcmp(argv[i], "-n")==0) {
		N = atol(argv[++i]) * 1024;
	}
	else if (strcmp(argv[i], "-s")==0) {
		MIN_SORT_SIZE = atol(argv[++i]);
	}
	else if (strcmp(argv[i], "-m")==0) {
		MIN_MERGE_SIZE = atol(argv[++i]);
```

```
	}
#ifdef _OPENMP
	else if (strcmp(argv[i], "-c")==0) {
		CUTOFF = atoi(argv[++i]);
	}
#endif
	else {
#ifdef _OPENMP
		fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE] -c CUTOFF\n", argv[0]);
#else
		fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
#endif
		fprintf(stderr, "	-n to specify the size of the vector
(in Kelements) to sort (default 32768)\n");
		fprintf(stderr, "	-s to specify the size of the vector
(in elements) that breaks recursion in the sort phase (default 1024)\n");
		fprintf(stderr, "	-m to specify the size of the vector
(in elements) that breaks recursion in the merge phase (default 1024)\n");
#ifdef _OPENMP
		fprintf(stderr, "	-c to specify the cut off recursion
level to stop task generation in OpenMP (default 16)\n");
#endif
		return EXIT_FAILURE;
	}
	}

	fprintf(stdout,
"****************************************************************************
**************\n");
	fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld, MIN_MERGE_SIZE=%ld\n", N/1024, MIN_SORT_SIZE,
MIN_MERGE_SIZE);
#ifdef _OPENMP
	fprintf(stdout, "Cut-off level: CUTOFF=%d\n",
CUTOFF);
```

```
	fprintf(stdout, "Number of threads in OpenMP:
OMP_NUM_THREADS=%d\n", omp_get_max_threads());
#endif
	fprintf(stdout,
"**************************************************************************
**************\n");

	T *data = malloc(N*sizeof(T));
	T *tmp = malloc(N*sizeof(T));

	double stamp;
	START_COUNT_TIME;

	initialize(N, data);
	clear(N, tmp);

	STOP_COUNT_TIME("Initialization time in seconds");

	START_COUNT_TIME;

	#pragma omp parallel
	#pragma omp single
	multisort(N, data, tmp, d);

	STOP_COUNT_TIME("Multisort execution time");

	START_COUNT_TIME;

	check_sorted (N, data);

	STOP_COUNT_TIME("Check sorted data execution time");

	fprintf(stdout, "Multisort program finished\n");
	fprintf(stdout,
"**************************************************************************
**************\n");
```

```
#ifdef _OPENMP
	TRACE_END_MARK;
#endif

	return 0;
}
```

4 Shared-memory

```C/C++
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "omp.h"
#include "fake.h"

#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);

// N and MIN must be powers of 2
long N;
```

```
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;
int CUTOFF;
int d;

#define T int

void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start,
long length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length) {
	if (length < MIN_MERGE_SIZE*2L) {
	// Base case
	basicmerge(n, left, right, result, start, length);
	} else {
	// Recursive decomposition
	#pragma omp task
	merge(n, left, right, result, start, length/2);
	#pragma omp task
	merge(n, left, right, result, start + length/2, length/2);
	#pragma omp taskwait
	}
}

void multisort(long n, T data[n], T tmp[n]) {
	if (n >= MIN_SORT_SIZE*4L) {
	// Recursive decomposition
	#pragma omp task depend(out:data[0])
	multisort(n/4L, &data[0], &tmp[0]);
	#pragma omp task depend(out:data[n/4L])
	multisort(n/4L, &data[n/4L], &tmp[n/4L]);
	#pragma omp task depend(out:data[n/2L])
	multisort(n/4L, &data[n/2L], &tmp[n/2L]);
	#pragma omp task depend(out:data[3L*n/4L])
```

```
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        #pragma omp taskwait

        #pragma omp task depend(in:data[0], data[n/4L]) depend(out:tmp[0])
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        #pragma     omp     task     depend(in:data[n/2L],     data[3L*n/4L])
depend(out:tmp[n/2L])
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        #pragma omp taskwait

        #pragma omp task depend(in:tmp[0], tmp[n/2L]) depend(out:data[0])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        #pragma omp taskwait
        } else {
        // Base case
        basicsort(n, data);
        }
}

static void initialize(long length, T data[length]) {
    long i;
    for (i = 0; i < length; i++) {
        if (i==0) {
                data[i] = rand();
        } else {
                data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
}

static void clear(long length, T data[length]) {
    long i;
    for (i = 0; i < length; i++) {
        data[i] = 0;
    }
}
```

```
void check_sorted(long n, T data[n])
{
    int unsorted=0;
    for (int i=1; i<n; i++)
        if (data[i-1] > data[i]) unsorted++;
    if (unsorted > 0)
        printf ("\nERROR: data is NOT properly sorted. There are %d unordered
positions\n\n",unsorted);
}

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

#ifdef _OPENMP
    TRACE_START_MARK;
#endif



    /* Defaults for command line arguments */
    /* Important: all of them should be powers of two */
    N = 32768 * 1024;
    MIN_SORT_SIZE = 1024;
    MIN_MERGE_SIZE = 1024;
    CUTOFF = 50;
    d = 0;

    /* Process command-line arguments */
    for (int i=1; i<argc; i++) {
        if (strcmp(argv[i], "-n")==0) {
              N = atol(argv[++i]) * 1024;
        }
        else if (strcmp(argv[i], "-s")==0) {
              MIN_SORT_SIZE = atol(argv[++i]);
        }
        else if (strcmp(argv[i], "-m")==0) {
              MIN_MERGE_SIZE = atol(argv[++i]);
        }
```

```
#ifdef _OPENMP
        else if (strcmp(argv[i], "-c")==0) {
                CUTOFF = atoi(argv[++i]);
        }
#endif
        else {
#ifdef _OPENMP
                fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE] -c CUTOFF\n", argv[0]);
#else
                fprintf(stderr, "Usage: %s [-n vector_size -s MIN_SORT_SIZE -m
MIN_MERGE_SIZE]\n", argv[0]);
#endif
                fprintf(stderr, "          -n to specify the size of the
vector (in Kelements) to sort (default 32768)\n");
                fprintf(stderr, "          -s to specify the size of the
vector (in elements) that breaks recursion in the sort phase (default
1024)\n");
                fprintf(stderr, "          -m to specify the size of the
vector (in elements) that breaks recursion in the merge phase (default
1024)\n");
#ifdef _OPENMP
                fprintf(stderr, "         -c to specify the cut off recursion
level to stop task generation in OpenMP (default 16)\n");
#endif
                return EXIT_FAILURE;
        }
    }

                                                        fprintf(stdout,
"******************************************************************************
**************\n");
         fprintf(stdout, "Problem size (in number of elements): N=%ld,
MIN_SORT_SIZE=%ld,     MIN_MERGE_SIZE=%ld\n",     N/1024,     MIN_SORT_SIZE,
MIN_MERGE_SIZE);
#ifdef _OPENMP
```

```
    fprintf(stdout, "Cut-off level:                                CUTOFF=%d\n",
CUTOFF);
    fprintf(stdout, "Number of threads in OpenMP:
OMP_NUM_THREADS=%d\n", omp_get_max_threads());
#endif
                                                                   fprintf(stdout,
"********************************************************************************
**************\n");

    T *data = malloc(N*sizeof(T));
    T *tmp = malloc(N*sizeof(T));

    double stamp;
    START_COUNT_TIME;

    initialize(N, data);
    clear(N, tmp);

    STOP_COUNT_TIME("Initialization time in seconds");

    START_COUNT_TIME;

    #pragma omp parallel
    #pragma omp single
    multisort(N, data, tmp);

    STOP_COUNT_TIME("Multisort execution time");

    START_COUNT_TIME;

    check_sorted (N, data);

    STOP_COUNT_TIME("Check sorted data execution time");

    fprintf(stdout, "Multisort program finished\n");
```

```
                                                                    fprintf(stdout,
"********************************************************************************
**************\n");

#ifdef _OPENMP
    TRACE_END_MARK;
#endif

    return 0;
}
```