Programming Parallel Embedded and Consumer Applications in OpenMP Superscalar

Michael Andersch
TU Berlin
Einsteinufer 17
10587 Berlin
andersch@cs.tu-berlin.de

Chi Ching Chi
TU Berlin
Einsteinufer 17
10587 Berlin
cchi@cs.tu-berlin.de

Ben Juurlink
TU Berlin
Einsteinufer 17
10587 Berlin
juurlink@cs.tu-berlin.de

Abstract

In this paper, we evaluate the performance and usability of the parallel programming model OpenMP Superscalar (OmpSs), apply it to 10 different benchmarks and compare its performance with corresponding POSIX threads implementations.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

General Terms Algorithms, Design, Measurement, Performance

Keywords OpenMP Superscalar, OmpSs, Embedded, Consumer

1. Introduction

OpenMP Superscalar (OmpSs) is a novel task-based parallel programming model which extends the OpenMP programming model with the StarSs [3] task directives. In OmpSs, programs are parallelized by annotating functions as tasks using the `omp task input output inout` pragmas. When these functions are called, they are added to a task graph instead of directly being executed. The task dependencies are resolved at runtime, using the input/output specification of the function arguments. Once all input dependencies of a task are resolved, it is ready to be executed.

In the past, OmpSs has mainly been used to parallelize HPC applications [2]. In this paper we will investigate and summarize the usability and performance of OmpSs for embedded and consumer applications.

This paper is organized as follows: Section 2 discusses our methodology. In Section 3 we investigate the expressiveness of OmpSs using pipelining in H.264 as a case study. In Section 4 the performance results of the ten benchmarks are presented. Finally, in Section 5, conclusions are drawn.

2. Methodology

Evaluating parallel programming models is different from evaluating processor architectures. Parallel programming models not only target good performance, but also must offer the right abstraction to the programmer. Therefore, it is necessary to investigate both the usability and performance of a parallel programming model to evaluate its overall quality.

The usability of a programming model is a subjective measure that differs from programmer to programmer. To provide the programmer with the necessary information to be able to form his/her own opinion about the usability, we conducted studies on numerous qualitative aspects, such as general expressiveness or the toolchain, of programming in OmpSs, of which we show H.264 pipeline parallelization as an example.

To evaluate the performance of OmpSs, we have created a benchmark suite to evaluate parallel programming models. The suite contains 10 C/C++ benchmarks (shown in the first column of Table 1) that cover a wide range of embedded and consumer application domains. The benchmarks are classified as kernels, workloads, or applications, based on their code size and parallelization complexity. For each benchmark a sequential, Pthreads, and OmpSs implementation have been developed. For comparability the Pthreads and OmpSs variants exploit the same parallelism.

To provide meaningful results not only for contemporary, but also for future multi-core systems, it is necessary to extend the benchmarking process beyond the core counts of what current off-the-shelve CMPs can offer. To achieve this, we use a 4-socket cc-NUMA machine with 32 cores in total for the performance evaluation.

3. Case Study: Parallelizing H.264 Decoding in OmpSs

The H.264 decoder pipeline in our design consists of 5 pipeline stages. In the read stage the bitstream is read from the disk and parsed into separated frames. In the parse stage the headers of the frame are parsed and a Picture Info entry in the Picture Info Buffer is allocated. The entropy decode (ED) stage performs a lossless decompression by extracting the syntax elements for each macroblock in the frame. The macroblock reconstruction stage allocates a picture in the Decoded Picture Buffer and reconstructs the picture using the syntax elements and motion vectors. The output stage reorders and outputs the decoded pictures either to an output file or the display.

In contrast to other task-based programming models, such as Cilk++ and OpenMP, pipeline parallelism can be easily expressed in OmpSs, because in OmpSs tasks can be spawned before their dependencies have been resolved [4, 5]. Listing 1 presents slightly simplified code of the pipelined main decoder loop using OmpSs pragmas. A task is created for each pipeline stage in each loop iteration. For correct pipelining of the tasks, it is required that all tasks in iteration *i* are executed in-order. To accomplish this, each task in the same iteration is linked to the previous task in the same iteration via one or more input and output/inout pairs. Additionally, task *T* of iteration *i* must be completed before the instance of the same task *T* in iteration *i+1* is started. To accomplish this, each task has a context structure parameter that is annoted as inout, e.g., ReadContext *rc, NalContext *nc, EntropyContext *ec, etc.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

```
EncFrame frm[N]; Slice slice[N];
H264Mb *ed_bufs[N]; Picture pic[N];
int k=0;
while(!EOF){
#pragma omp task inout(*rc) output(*frm)
  read_frame_task(rc, &frm[k%N]);
#pragma omp task inout(*nc,*frm) output(*s)
  parse_header_task(nc,&slice[k%N], &frm[k%N]);
#pragma omp task inout(*nc,*frm) output(*s)
  entropy_decode_task(ec,&slice[k%N], &frm[k%N],
    ed_bufs[k%N]);
#pragma omp task inout(*rc) input(*s,*mbs) output(*pic)
  reconstruct_task (rc,&slice[k%N],ed_bufs[k%N],
    &pic[k%N]);
#pragma omp task inout(*oc) input(*pic)
  output_task(oc,&pic[k%N]);
  k++;
#pragma omp taskwait on (*rc)
}
```

Listing 1. Pipelining the main decoder loop using OmpSs pragmas

Three additional important observations regarding the pipelining implementation can be made. First, the `taskwait on` pragma ensures that the read task has been performed before evaluating the while loop condition. This is necessary to prevent tasks from being added after the EOF has been reached.

Second, more importantly, some inputs and outputs of each task are read from/written to an entry of a circular buffer of size *N*. This eliminates the WAR and WAW hazards that would have occurred if the same entry is used in each iteration, which would eliminate all the parallelism. OmpSs does not support automatic renaming and, therefore, this manual renaming method is required.

Third, the Picture Info Buffer (PIB) and Decoded Picture Buffer (DPB) structures are not passed in any argument and, thus, are not considered for dependence checking. The dependencies to these buffer entries are purposely hidden from the OmpSs task specifications, because we cannot predict which buffers entries will be available at the time the task is spawned. This can only be determined when the task is executed.

To fetch and release the buffer entries in a thread-safe way, `omp critical` pragmas are used in the task bodies around the fetch and release statements to protect accesses to the PIB and DPB.

4. Quantitative Evaluation

In Table 1 the speedups of the OmpSs variants over the Pthreads variants are shown for each benchmark and core count. Overall, five benchmarks are faster with OmpSs and four with Pthreads. The largest gains are observed for the c-ray, rgbcmy, and ray-rot benchmarks. The largest loss is observed for h264dec.

In the rgbcmy benchmark multiple iterations are performed to stabilize the execution time, with a task/thread barrier separating each iteration. The absolute time for one iteration, however, is short with less than 20ms on 16 cores. For this benchmark, the OmpSs variant is able to scale better at higher core counts because it employs a polling task barrier instead of the more expensive blocking thread barrier.

In the ray-rot benchmark the output of the c-ray kernel is the input of the rotate kernel. For this benchmark OmpSs performs better than Pthreads, because the runtime scheduler places dependent tasks on the same core. Scheduling tasks that have an input output relation back-to-back on the same core improves cache locality. Interestingly, due to this locality advantage, the speedups for the combined ray-rot workload exceed the product of the speedups of the individual c-ray and the rotate kernel.

The largest performance difference between Pthreads and OmpSs occurs for the h264dec benchmark. Increasing the task granularity is necessary to improve the overall performance of OmpSs. Grouping the tasks, however, reduces the parallelism, which in turn limits the performance at higher core counts. In the Pthreads version of h264dec the synchronization is highly optimized using a line decoding strategy [1] and, therefore, grouping of tasks is not necessary.

Benchmark	1	8	16	24	32	Mean
c-ray	1.03	1.11	1.12	1.11	1.14	1.10
rotate	1.06	1.04	1.09	1.02	0.86	1.01
rgbcmy	1.02	0.98	1.14	1.40	1.53	1.19
md5	1.00	1.02	1.10	1.14	1.05	1.06
kmeans	0.91	0.87	1.30	0.95	0.88	0.97
ray-rot	1.02	1.10	1.65	1.46	1.20	1.27
rot-cc	1.00	1.06	1.17	1.14	1.04	1.08
streamcluster	0.93	0.84	0.91	0.99	0.99	0.93
bodytrack	0.98	0.99	1.05	0.97	1.00	1.00
h264dec	0.94	1.07	0.87	0.57	0.42	0.73
Mean	0.99	1.00	1.12	1.05	0.97	1.02

Table 1. Speedup factors and geometric means of OmpSs implementations over Pthreads implementations for each benchmark and core count.

Over the entire benchmark suite, OmpSs performs 2% better than Pthreads. At 1 and 8 cores the performance is very close, while at 16 and 24 cores OmpSs is slightly faster. At 32 cores OmpSs is slightly slower mainly due to the lower performance in the h264dec benchmark. Thus, we argue that performance wise OmpSs can compete with manual threaded solutions for the embedded and consumer benchmarks considered in this paper.

To be a true alternative for manual threading, however, OmpSs processes must be able to dynamically share resources with other processes. Currently, OmpSs programs use a static number of cores controlled by an environmental variable. Furthermore, because the runtime implements core communication/synchronization, e.g. task barriers, in a polling fashion for performance reasons, all used cores are always fully loaded even if there is insufficient work. This reduces overall system responsiveness and power efficiency when too many cores are used.

5. Conclusions

Our studies have shown that OmpSs is, while not yet production-ready, a viable alternative to established parallel programming models such as Pthreads. The expressiveness is sufficiently powerful to program common parallelism patterns such as pipelining and the performance is comparable to Pthreads implementations.

References

[1] C. C. Chi and B. Juurlink. A QHD-Capable Parallel H.264 Decoder. In *Proc. 25th Int. Conf. on Supercomputing*, 2011.

[2] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proc. Int. Conf. on Parallel Processing*, 2009.

[3] J. M. Perez, R. M. Badia, and J. Labarta. A Flexible and Portable Programming Model for SMP and Multi-cores. Technical report, BSC-UPC, 2007.

[4] A. Pop and A. Cohen. A Stream-Computing Extension to OpenMP. In *Proc. 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*, 2011.

[5] H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proc. 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.