Unrolling Loops Containing Task Parallelism

Roger Ferrer[1], Alejandro Duran[1], Xavier Martorell[1,2], and Eduard Ayguadé[1,2]

[1] Barcelona Supercomputing Center
Nexus II, Jordi Girona, 29, Barcelona, Spain
{*roger.ferrer,alex.duran*}*@bsc.es*
[2] Departament d'Arquitectura de Computadors
Edifici C6, Jordi Girona, 1-3, Barcelona, Spain
{*xavim,eduard*}*@ac.upc.edu*

Abstract. Classic loop unrolling allows to increase the performance of sequential loops by reducing the overheads of the non-computational parts of the loop. Unfortunately, when the loop contains parallelism inside most compilers will ignore it or perform a naïve transformation.
We propose to extend the semantics of the loop unrolling transformation to cover loops that contain task parallelism. In these cases, the transformation will try to aggregate the multiple tasks that appear after a classic unrolling phase to reduce the overheads per iteration.
We present an implementation of such extended loop unrolling for OpenMP tasks with two phases: a classical unroll followed by a task aggregation phase. Our aggregation technique covers the special cases where task parallelism appears inside branches or where the loop is uncountable.
Our experimental results show that using this extended unroll allows loops with fine-grained tasks to reduce the overheads associated with task creation and obtain a much better scaling.

1 Introduction and Motivation

In the high performance computing community we always strive to squeeze a bit more of performance by reducing the time that the program spends not doing useful computation.

As such, many compiler transformations[12, 6] are aimed at reducing some of these overheads in both sequential and parallel applications. One classical transformation is loop unrolling[1]. Loop unrolling applied to a sequential loop allows to reduce the overhead by reducing the number of condition checks and branches of the loop. Unfortunately if the loop is parallel or contains parallel parts, compilers will just ignore the potential unrolling or perform just the classical transformation which will not reduce the main overheads of the loop.

We present an extended semantic for loop unrolling where unrolling the loop should try to reduce all possible overheads of multiple iterations to those of a single iteration. In this work, we concentrate in reducing the overheads of loops that contain task parallelism inside them. This strategy resembles the one used by vectorizing compilers[2] where scalar computation in loops is aggregated into vector operations after applying a clasical unrolling.

Tasks parallelism is an easy-to-understand concept and may lead programmers to use it extensively in their codes including loops. While this allows to maximize the available parallelism in the application it can also yield poor performance if the tasks are fine-grained because of the associated overheads.

Our compiler allows the user to annotate a loop for unrolling by means of a pragma directive. We have extended the unrolling transformation to match the previous definition so it tries to reduce the overheads associated with creation of tasks inside the loop. The loop is first unrolled as usual and then the tasks inside it are aggregated to reduce the overheads. Our approach is able to cope with cases where there is more than one single code path in the loop body (e.g., branches). As an automatic benefit of this support, our unrolling can also be applied with uncountable loops (something not very useful with classical unroll but that can still yield benefits in our case).

Our goal is not deciding when this transformation should be used, as other proposals for high-level transformations[6, 3, 4], we make it available for programmers (or other compiler phases) so they can choose to apply it in those cases where it can help to reduce overheads. Certainly, this technique could be integrated with automatic approaches for task parallelism[9, 5, 7, 8].

Through this paper we will use the tasking model defined by the latest (3.0) OpenMP specification for the different examples but by no means are the transformations or results tailored to this language specificall. They are general enough to be applied to any task-parallel programming model.

2 Unrolling task parallelism

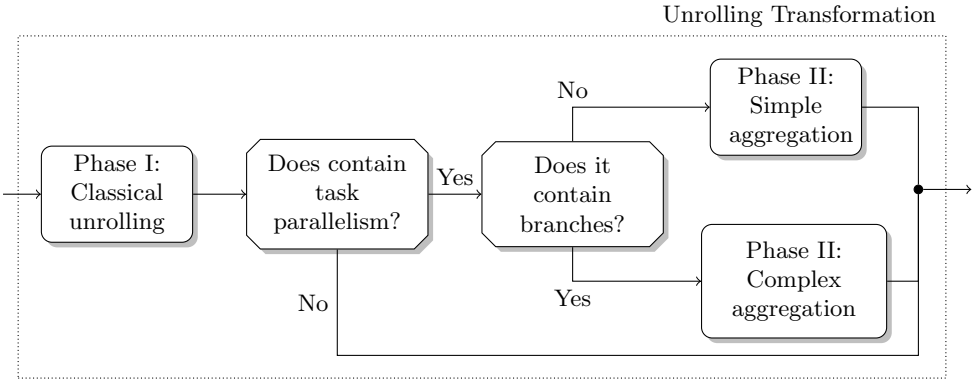

Fig. 1: Phases scheme to unroll loops

Unrolling countable loops is beneficial since it reduces the overhead related to the end of loop test and branching. Thus, unrolling can be understood as a

way of increasing the granularity of each iteration by doing more work between the unavoidable branches of the loop.

In this section we describe our extension of unrolling to fit this concept of overhead reduction for loops that contain task parallelism. First, we explain the the basic strategy for simple loops. Then we describe how we tackle more complex cases like loop bodies with conditionally created task parallelism and uncountable loops.

Fig. 1 depicts the different stages of our unrolling strategy. Consider the simple case in Fig. 2a where a loop creates a task at every iteration[3].

If the original loop contained task parallelism, after *classical unrolling* we would have a set of replicated tasks (see Fig. 2b). We remove each of these tasks and instead we add a new aggregated task containing the code of the replicated tasks along with all their inputs (see Fig. 2c). This aggregated task is coarser and its creation cost is on par of the original tasks: overall overhead has been reduced.

```
1 for (i = 0; i < N; i++)
2 #pragma omp task firstprivate(i)
3     f(i);
```

(a) Before unroll

```
1 for (i = 0; i <= N - UF; i += UF)
2 {
3 #pragma omp task firstprivate(i)
4     f(i);
5 #pragma omp task firstprivate(i)
6     f(i + 1);
7 #pragma omp task firstprivate(i)
8     f(i + 2);
9 ...
10 #pragma omp task firstprivate(i)
11     f(i + UF - 1);
12 }
13 for (; i < N; i++) // Epilogue
14 {
15 #pragma omp task firstprivate(i)
16     f(i);
17 }
```

(b) *Classical unroll* applied to Fig. 2a. `UF` stands for $unrollfactor$

```
1 for (i = 0; i <= N - UF; i += UF)
2 {
3 #pragma omp task firstprivate(i)
4   {
5     f(i);
6     f(i + 1);
7     f(i + 2);
8     ...
9     f(i + UF - 1);
10  }
11 }
12 for (; i < N; i++) // Epilogue
13 {
14 #pragma omp task firstprivate(i)
15     f(i);
16 }
```

(c) Simple aggregation applied to Fig. 2a. `UF` stands for $unrollfactor$

Fig. 2: Simple loop containing a parallel task.

Aggregation of tasks is feasible because parallel tasks can be moved in the code as long as their dependencies are preserved. For the particular case of OpenMP, `firstprivate` variables can be understood as the input dependencies

[3] In OpenMP[10], parallel tasks are created using `#pragma omp task`.

of parallel tasks. There are no explicitly output dependencies in OpenMP (like it happens in other parallel programming models like SMP Superscalar[11]), only a `taskwait` directive for waiting all created tasks that have not ended. Therefore, when moving tasks in OpenMP, a `taskwait` cannot be crossed. It is also worth mentioning that, although the examples presented here have only one task in the loop body, nothing prevents to apply this transformation to loop bodies with more than one task.

The simple aggregation can only be used when the program does not create tasks conditionally. If we aggregate these tasks, we have to ensure that the aggregated task only executes the code of the tasks that were actually created by the program.

```
1 for (i = 0; i < N; i++)
2 {
3     if (cond(i))
4 #pragma omp task firstprivate(i)
5         f(i);
6 }
```

(a) Loop with conditional task

```
1 for (i = 0; i <= N - UF; i += UF)
2 {
3     if (cond(i))
4 #pragma omp task firstprivate(i)
5         f(i);
6     if (cond(i + 1))
7 #pragma omp task firstprivate(i)
8         f(i + 1);
9     if (cond(i + 2))
10 #pragma omp task firstprivate(i)
11         f(i + 2);
12 ...
13     if (cond(i + UF - 1))
14 #pragma omp task firstprivate(i)
15         f(i + UF - 1);
16 }
17 // Epilogue omitted
```

(b) Loop with conditional task *classically unrolled*. `UF` stands for $unroll factor$

Fig. 3: Conditional task creation.

Consider the example in Fig. 3a where a task is created conditionally. *Classical unroll* would create a set of tasks along with all the conditionals (see Fig. 3b). To aggregate these tasks we have devised a simple strategy that we call *predication*.

Predication is a simple way to aggregate conditionally created tasks using guards. There is one guard per originally created task. The aggregated task not only receives the inputs of each task, as it happens in a simple aggregation, but also the guards. These guards are used in the aggregated task to enable or disable the code of the conditional tasks.

Fig. 4 shows the predication strategy applied to the loop of Fig. 3b. Implementing the predicate logic is very simple and keeps the original structure of the code. Predication information can be packed into bits so the amount of data needed for the predicates can be very small. Predicates can also benefit

from architectures with predication in their instruction sets. As a very simple optimization, if no tasks are created in the whole body of the unrolled loop no aggregated task is created either.

```
1 struct _guard_1_1
2 {
3     char _task_guard_0 : 1;
4     char _task_guard_1 : 1;
5     ...
6     char _task_guard_{UF-1} : 1;
7 };
8 for (i = 0; i <= N - UF; i += UF)
9 {
10    struct _guard_1_1 _g_1 = {0};
11    char _iteration_guard = 0;
12    if (cond(i + 0))
13        _iteration_guard = _g_1._task_guard_0 = 1;
14    // ... Similar cases for i + 1, i + 2, ...
15    if (cond(i + UF - 1))
16        _iteration_guard = _g_1._task_guard_{UF-1} = 1;
17    if (_iteration_guard)
18    {
19#pragma omp task firstprivate(i, _g_1)
20        {
21            if (_g_1._task_guard_0)
22                f(i + 0);
23            // ... Similar cases for
24            // _task_guard_1, _task_guard_2, ...
25            if (_g_1._task_guard_{UF-1})
26                f(i + UF - 1);
27        }
28    }
29 }
30 // ... Epilogue omitted, likewise the one in Figure 2b
```

Fig. 4: Loop in Fig. 3a unrolled using predication. `UF` stands for *unroll factor*

The aggregated task keeps the order of creation in the original code, so if the code had the serial execution optimized (e.g., for locality) this is preserved in the aggregated task.

Predication links the unrolled iteration with the aggregation amount. This is a drawback because, in a worst-case scenario, every unrolled iteration could end creating just a task so the aggregated task would end executing one task. In this case, the overhead for task creation would be similar to the original code.

Uncountable loops, like `while` loops, can also benefit of the techniques shown here. Although unrolling this kind of loops is usually of little use, since it does not reduce the number of branches, parallel tasks in the loop body can still be aggregated using the techniques presented here leading to a reduction of the overhead caused by task parallelism creation.

3 Evaluation

We extended our Mercurium C/C++ source-to-source compiler, which already supports OpenMP, to implement the unrolled task aggregation. This transformation was performed in the compiler in an earlier phase than OpenMP, so the latter phase implements the OpenMP semantics. Our compiler currently targets our NANOS 4[13] runtime.

A quite common case is having a data structure that needs to be traversed and some operation to be applied to all its elements. Typically, each element can be processed in parallel but because in many cases the operations are very short the resulting parallelism is too fine-grained to be exploited effectively. We have used a synthetic benchmark that tries to model this situation by implementing a linked list traversal. For each element an operation of some `cost` is applied. Fig. 5 shows the OpenMP implementation with *unroll* being applied to the loop. By adjusting the number of elements of the list and the `cost` of the `fill` function we can test different granularity scenarios. Our objective in this experiment was to discover if using our unrolling method will help to scale applications with this kind of fine-grain parallelism.

All results depict speedups relative to a serial execution without OpenMP. In order to see the real improvement when no unrolling is applied, we used a *Baseline* execution using OpenMP but without unrolling.

We executed the benchmarks in a SGI Altix 4700 with 128 Itanium 2 processors using a *cpuset* configuration that avoids interferences with other threads in the system. For the purpose of this evaluation we will not use more than 32 processors. The backend compiler used by Mercurium source-to-source compiler was `gcc` 4.1.

```
1 std::list<int> l;
2 fill_list(l, length);
3
4 #pragma omp parallel
5 {
6 #pragma omp  single
7 #pragma hlt unroll factor(N)
8   for (list<int>::iterator it = l.begin(); it != l.end(); it++ )
9 #pragma omp task untied firstprivate(it, cost)
10     fill( *it, cost);
11 }
```

Fig. 5: List traversal code

We have evaluated several pairs of `cost` and `length` along with several unrolling factors for *predication*. We show in Fig. 6 some results for a number of different task sizes (the number of elements was chosen so the total amount of work in each case was roughly the same). We can see that for task sizes of around 200 μs the base implementation obtains good speed-ups, but, even so, unrolling the loop with a small factor yields close to perfect speed-up. When we reduce

the task size to a tenth ($\sim$27μs) of that the base implementation performance sinks while the unrolled versions still obtain roughly the same performance when unrolling more than 64 iterations. Reducing it even further to $\sim$2μs the limits of our technique start to show and only scales up to 8 threads. The problem that limits the scalability is that the thread aggregating tasks cannot generate them fast enough to keep all the threads working. Careful tuning of the aggregation code could increase this limit further.

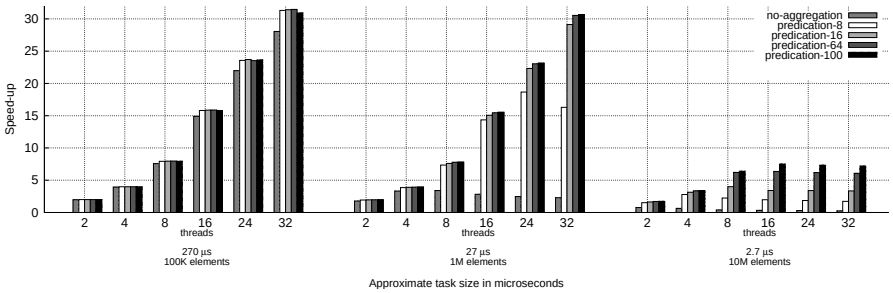

Fig. 6: Lists synthetic benchmark speedups

4 Conclusions and Future Work

In this work, we have presented an unroll technique that extends the semantic of the classical unroll transformation to cope in a non-naïve way with loops that contain task parallelism. The technique is generic enough to handle loop bodies where tasks are conditionally created or multiple tasks exist in the body.

We have seen from the evaluation that for some scenarios where the base runtime was not able to scale due to too fine grained parallel tasks, it is possible to improve the scalability thanks to our extended unrolling.

Another interesting work is devising ways in which the compiler can determine when to automatically apply this transformation instead of being directed by the programmer. Besides determining whether the loop would benefit of the unrolling the compiler should also be able to determine the optimal factor of unrolling to apply to the loop.

Acknowledgments

This research was supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC-2 Network of Excellence (contract no. FP7/IST-217068) and the Mare Incognito project under the BSC-IBM collaboration agreement.

References

1. F.E. Allen and J. Cocke. A Catalogue of Optimizing Transformations. *Design and Optimization of Compilers*, pages 1–30, 1972.
2. Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.
3. Chun Chen, Jacqueline Charme, and Mary Hall. CHiLL: A Framework for Composing High-Level Loop Transformations. 2008.
4. Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Mara Garzarn, David Padua, and Keshav Pingali. *Languages and Compilers for Parallel Computing*, chapter A Language for the Compact Representation of Multiple Program Versions, pages 136–151. Springer-Verlag, 2006.
5. A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.
6. S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.
7. Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih wei Liao, and Monica S. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler, 1995.
8. Kazuhisa Ishizaka, Motoki Obata, and Hironori Kasahara. Coarse-Grain Task Parallel Processing Using the OpenMP Backend of the OSCAR Multigrain Parallelizing Compiler. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 457–470, London, UK, 2000. Springer-Verlag.
9. C. McCreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. 1989.
10. OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2008.
11. Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A Flexible and Portable Programming Model for SMP and Multi-cores. Technical report, Barcelona Supercomputing Center-Centro Nacional de Supercomputacin, 2007.
12. William Pugh. Uniform Techniques for Loop Optimization. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 341–352, New York, NY, USA, 1991. ACM.
13. Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for OpenMP Tasks in Nanos v4. In *CAS Conference 2007*, October 2007.