Adaptive Granularity Control in Task Parallel Programs Using Multiversioning

Peter Thoman, Herbert Jordan, and Thomas Fahringer

University of Innsbruck, Institute of Computer Science
Technikerstrasse 21a, 6020 Innsbruck, Austria
{petert,herbert,tf}@dps.uibk.ac.at

Abstract. Task parallelism is a programming technique that has been shown to be applicable in a wide variety of problem domains. A central parameter that needs to be controlled to ensure efficient execution of task-parallel programs is the granularity of tasks. When they are too coarse-grained, scalability and load balance suffer, while very fine-grained tasks introduce execution overheads.

We present a combined compiler and runtime approach that enables automatic granularity control. Starting from recursive, task parallel programs, our compiler generates multiple versions of each task, increasing granularity by task unrolling and subsequent removal of superfluous synchronization primitives. A runtime system then selects among these task versions of varying granularity by tracking task demand.

Benchmarking on a set of task parallel programs using a work-stealing scheduler demonstrates that our approach is generally effective. For fine-grained tasks, we can achieve reductions in execution time exceeding a factor of 6, compared to state-of-the-art implementations.

Keywords: Compiler, Runtime System, Parallel Computing, Task Parallelism, Multiversioning, Recursion.

1 Introduction

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [1]. While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. A central feature of every task-based parallel program that significantly affects both efficiency and scalability is *task granularity* [8]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks.

Very fine-grained, short-running tasks lead to a loss in efficiency compared to sequential execution due to the runtime overhead associated with generating and launching a task, as well as synchronizing its completion with other tasks in the system. On the other hand, coarse-grained, long-running tasks minimize overhead, but are hard to schedule effectively and may therefore fail to scale well on large parallel systems. Previous work in this area has focused mostly on

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 164–177, 2013.

runtime systems or user-controlled cutoffs to manage granularity (see Section 5). Conversely, we propose an approach that combines a multiversioning compiler with a runtime system which adaptively selects from the generated versions. Our goal is to maximize efficiency by increasing task granularity – and thus decreasing overheads – without negatively affecting load balance or scalability.

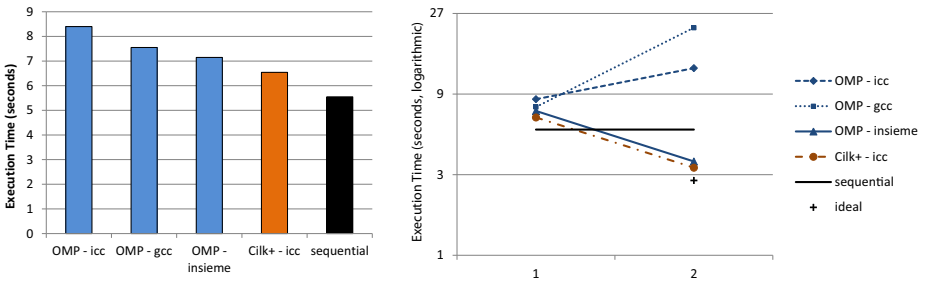
We implemented our method for OpenMP [17] tasks within the Insieme compiler and runtime system [11], but the idea is equally applicable to any other task parallel language. Our concrete contributions are the following:

- A compile-time multiversioning transformation that generates a set of task implementations of increasing granularity by recursive *task unrolling* and subsequent elimination of superfluous synchronization primitives. This transformation is applicable to both simple recursion and N-ary mutual recursion.
- A runtime heuristic for the dynamic adaptation of granularity based on the concept of *task demand*, which automatically choses the code version to execute at each task spawning point.
- Evaluation and analysis of the performance of our method on a number of well-known task parallel benchmarks. We compare with other OpenMP implementations, our own implementation without the multiversioning optimization and Cilk [2] versions which represent the state of the art in fine-grained task parallelism.

The remainder of this paper is structured as follows. In Section 2 we provide some initial results that motivated our work. We then describe our method in detail in Section 3 and evaluate its performance in Section 4, followed by an overview of related work in Section 5. Finally, Section 6 concludes the paper.

2 Motivation

Figure 1(a) shows single-threaded execution times measured for the Barcelona OpenMP Tasks Suite (BOTS) [7] N-Queens benchmark with $N = 13$. For details on the hardware, compiler versions and programs used refer to Section 4.



(a) Single-threaded (b) Scaling from 1 to 2 threads

Fig. 1. Initial Experiments, N-Queens $N = 13$

The lowest execution time amongst the OpenMP versions is achieved by our compiler and runtime system (Insieme), however, this time is still 28% higher than purely sequential execution. Even the Cilk version, while more efficient than any OpenMP implementation, is 19% slower than the sequential version. Our multiversioning method is designed to address this inefficiency. Throughout this paper, when we refer to *inefficient* execution, we mean execution which takes longer than executing purely sequential code (assuming perfect scaling).

Note that the OpenMP runtime systems of ICC [12] and GCC [19] perform special case handling when only a single worker thread is used. This is visible in Figure 1(b), which shows their performance degrading when switching from one to two threads. Further experiments in Section 4 confirm this behavior, with scaling starting after some initial performance degradation when activating multi-threaded execution. The OpenMP version compiled with Insieme and the Cilk version do not suffer from this issue, however they still induce a relative overhead of about 20% compared to ideal linear scaling from the sequential version. We identified the following potential causes for this inefficiency:

1. Task generation overhead. This includes generating a task structure, populating it with values and enqueuing it.
2. Synchronization primitive overhead (e.g. `taskwait`). At the very least, this involves keeping track of all the subtasks launched by each task, and signaling when they are complete.
3. Task library calls. The runtime methods required for tasking are generally implemented in a separate library, and the overhead for their invocation is incurred even if they perform no actual work.
4. Non-inlineable, indirect program function calls. Since the program function implementing a given task needs to be called by the tasking library, a pointer to it is usually passed to the library function. Even if the runtime library decides to directly execute the call, this prevents the benefits – improved instruction scheduling and a reduction in overhead – associated with inlining.

Issues 1 and 2 can be mitigated by a pure runtime approach, e.g. the runtime library can dynamically decide whether to generate a full task structure or directly call the task function. This method is usually referred to as lazy task creation [14]. However, the basic overhead of library function calls (issue 3) and the fact that indirectly called functions in the original program can not be inlined (issue 4) can not be changed at runtime and need to be handled at compile time. This limitation of pure runtime systems motivates our compiler-aided multiversioning approach.

All four potential causes for inefficient execution identified above are directly related to and influenced by the granularity of tasks. The more often individual tasks are generated and synchronized, the higher the impact of the associated overheads on execution time. However, simply increasing the granularity of all tasks is not a solution: such an approach will lead to load imbalance, increasing the probability of workers idling. Therefore, our goal is the generation of different implementations for each task.

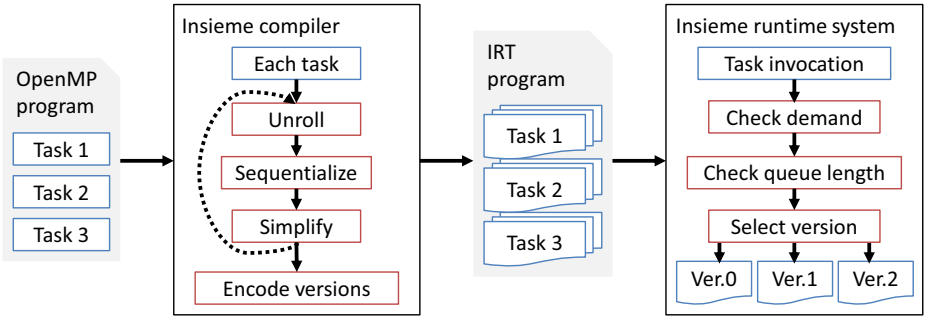


Fig. 2. Overview of our Method

3 Method

Figure 2 provides an overview of our proposed method. Starting from an OpenMP program with parallel tasks, our compiler generates an IRT (Insieme Runtime) program in which multiple different implementation versions of each task are encoded. During execution of the program, whenever a specific task is invoked, the Insieme runtime system selects and launches a version of this task.

3.1 Compile-Time Multiversioning

During compilation our goal is to generate multiple versions of each parallel task, with varying granularity. As depicted in Figure 2 this involves a three step process, which may be applied multiple times to further increase the task size. The individual steps are as follows:

1. **Task unrolling**. Replaces each task invocation site with a direct call to the task function, which is subsequently inlined. This can be thought of as a context and parallelism-aware recursive function inlining step. The name *task unrolling* is adapted from Rugina's usage of *recursion unrolling* [18].
2. **Sequentialization**. This step focuses on identifying which synchronization primitives – if any – were rendered superfluous by the partial elimination of parallel task invocations due to task unrolling, and removing them. It is described in more detail below.
3. **Simplification**. The unrolling and sequentialization may have generated code that can be simplified by basic operations such as arithmetic simplification, constant propagation or dead code elimination. Thus, these are performed before any further processing.

The number of generated versions depends on the granularity of the initial tasks and the largest granularity desired. The versions are generated and encoded into the target program in the following order.

1. **Original**. The original version from the input program.
2. N **times unrolled versions**. Starting from $N = 1$. In these versions, only partial sequentialization is performed. Outer task spawning points are

removed, but the innermost spawning location is kept. This process is illustrated in detail in a code example in Figure 4, described below.

3. **Fully sequentialized version**. In this version all task spawning points are removed and replaced with plain function calls.

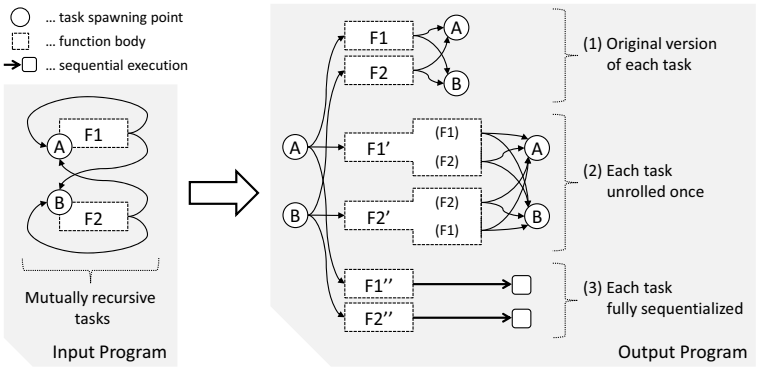


Fig. 3. Version Generation and Control Flow

Figure 3 illustrates the result of generating 3 versions for a mutually recursive task set consisting of two functions $F1$ and $F2$. The original program thus has two task spawning locations, A (which spawns $F1$) and B (spawning $F2$). To improve the clarity of the illustration, these task spawning points have been replicated in the figure, however they are still all referring to the same task.

Version (1) is identical to the original program, except that at each spawning point there is now a choice between 3 distinct implementations of each function. In version (2), consisting of $F1'$ and $F2'$, each recursive task invocation was unrolled once, forming tasks of increased granularity. Clearly, if this version is used, more work is performed between individual task invocations and interactions with the runtime library. Finally, version (3), comprising $F1''$ and $F2''$, is fully sequentialized. Once this version is invoked, no further parallel tasks will be spawned on this branch of the recursive descent.

Code Example. Figure 4 illustrates the effect of the steps taken during compilation to generate a task version that has been unrolled once. A pseudo-code formulation is used for reasons of clarity and size. It is C-like, but without the need for explicit type specification, and with two additional keywords: `spawn` implies the generation of a new parallel task (corresponding to `#pragma omp task untied`), while `merge_all` waits for the completion of all launched subtasks (equivalent to `#pragma omp taskwait`).

In (a), the original input code is shown. Moving on to (b), first-level task invocations are removed and replaced with in-place calls of the associated functions. Context-sensitive inlining of these calls results in (c). Finally, redundant applications of the `merge_all` operation are removed and arithmetic simplification is

applied. The final generated code for this version is listed in (*d*). This process can be repeated N times to generate increasingly larger task sizes.

After all the versions are generated, each version needs to be modified to enable runtime selection. Figure 5 contains the final code for the original version with task selection (*a*), the unrolled version as discussed previously (*b*) and a fully sequentialized version (*c*). The `pick` keyword implies a possible choice between semantically equivalent versions, which is to be deferred to the runtime system.

Partial Sequentialization. In most parallel programs there will be some superfluous synchronization statements after task unrolling. Since the execution has been partially sequentialized, instructions that wait for the completion of a task that was unrolled are no longer necessary and should be removed. The transformation eliminating unnecessary synchronization acts as follows on a task version T, effectively removing all `merge_all` operations for which there is no possibility of any task being spawned between them and a previous `merge_all`:

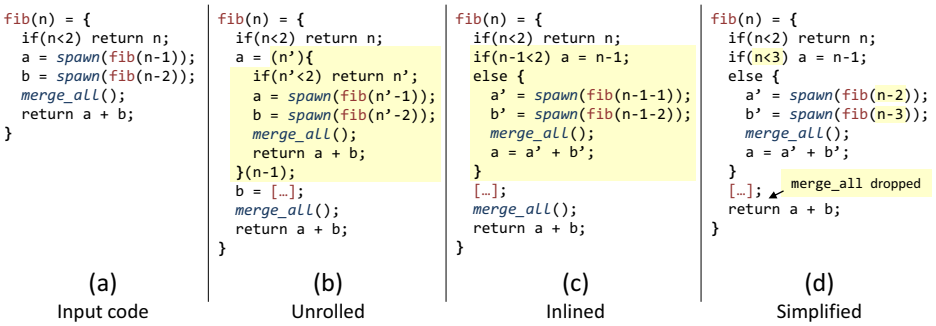


Fig. 4. Example task transformation - Fibonacci - Version generation

1. Determine the set S of all `merge_all` invocations in T.
2. For each `merge_all` $M \in S$:
   (a) Compute the set of all execution paths F from the entry point of T to M.
   (b) Reverse the paths in F.
   (c) If no path in F encounters a `spawn` before reaching a `merge_all`, remove M from T.

3.2 Runtime Version Selection

The previous section outlined how multiple versions with different granularities and trade-offs are generated in the compiler. This provides the runtime system with an opportunity of making a version choice every time a task is spawned. Making the wrong choice can result in not gaining the desired increase in efficiency, or, at worst, greatly diminishing parallelism – e.g. in case a fully sequentialized version is chosen too early. We considered the following design goals and observations when developing our version selection method:

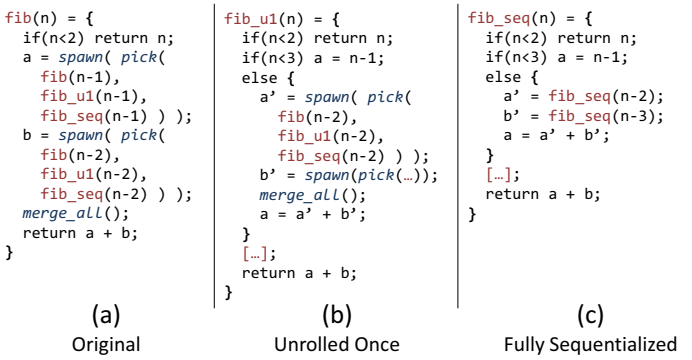


Fig. 5. Example task transformation - Fibonacci - Generated versions

- At the start of the program, the original (most fine-grained) version of the tasks should be used, since the parallelism available in the system is not yet fully leveraged and load-balancing is a priority.
- The impact of conservative behavior – i.e. using more fine-grained tasks – causes more gradual performance degradation than using tasks that are too coarse grained, potentially leading to some worker threads idling.
- The decision procedure needs to be simple and not introduce large overheads on its own, otherwise it could negate any benefits from multiversioning.
- The decision making process should be distributed – no new synchronization points between worker threads should be introduced to facilitate version selection.

Taking these points into account led to the development of a distributed version selection heuristic based on two parameters that are tracked for each individual worker thread. The first is *task demand*, which keeps track of other worker's unfulfilled attempts to steal tasks from the local worker. The second parameter is the *queue length* of each worker, or how many tasks it currently has available to be executed or stolen.

Task demand is tracked in a surprisingly simple, but effective, manner. The demand is stored as an integer which starts at a positive value equal to the maximum task queue length. Whenever a task is generated by a worker thread, it reduces its own task demand by 1. When a worker attempts to steal from another which has no tasks available, that target worker's demand value is reset to the starting value.

Our version selection heuristic is described in Figure 6. In conjunction with the demand tracking outlined above, it has the following desirable properties:

- Evaluating the selection function only takes a few dozen cycles, assuming that all the required values are cached.
- The way in which task demand is completely reset if any stealing operation fails, but is only reduced gradually during normal execution, mirrors the earlier observation about the negative performance impact of wrong granularity selection. It makes the expensive case of idle workers unlikely by reacting very strongly to failed stealing attempts.

`queue_length`	current queue length
`task_demand`	current task demand
`num_versions`	number of versions generated for current task
`MAX_QUEUE`	maximum queue length (fixed)
output:	$0 \Leftrightarrow$ original task
	$N = 1 \ldots$ `num_versions` $- 2 \Leftrightarrow$ unrolled N times
	`num_versions` $- 1 \Leftrightarrow$ fully sequentialized

```
1: version = num_versions − ⌈(task_demand/MAX_QUEUE) ∗ num_versions⌉
2: if version >= num_versions − 1 then
3:   if queue_length == MAX_QUEUE then
4:     return num_versions − 1
5:   end if
6:   return num_versions − 2
7: end if
8: return version
```

Fig. 6. Version Selection Algorithm

– Selecting the fully sequentialized version is a step that should only be taken after careful consideration, since it will prevent any further parallelism from being generated on this branch of the recursive descent. Therefore, the heuristic only takes this step if there has been no demand for additional tasks over a large number of spawn points *and* the queue is full.

The choice of the `MAX_QUEUE` parameter has an impact on the effectiveness of this approach. Experimental evaluation has shown that generally, a longer queue is beneficial on systems with a larger number of cores. For the evaluation in Section 4, `MAX_QUEUE` was set to 32.

4 Evaluation

In this section we will evaluate the performance impact of our optimization on multiple benchmark programs. Subsection 4.1 details our measurement methodology and the experimental setup used. We will perform an in-depth evaluation of one program in Subsection 4.2, and then proceed with an overview of the results of a number of other codes in order to provide a balanced overall impression.

4.1 Experimental Setup

For our experiments we used an Intel-based parallel system, incorporating 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. Table 1 summarizes the configuration of this system.

Table 1. Hardware and software platform for experimental evaluation

Sockets/ Cores	Cache			Software				
	L1d/i	L2	L3	OS	Kernel	GCC	ICC	Insieme
4/40	32K/32K	256K	30M	CentOS 6.3	2.6.32	4.6.3	12.1	g4614502

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. All experimental runs were repeated five times, and the median runtime is reported.

While the most important comparison for our evaluation is between our compiler with and without our multiversioning method, we also included the results obtained by other platforms to provide a reference for comparison. Table 1 includes the exact version number of the compilers used in these comparisons. ICC was used as the backend compiler for the Insieme source to source infrastructure, and its built-in Cilk Plus support was employed to compile Cilk programs. The optimization flag "`-O3`" was enabled for all calls to GCC and ICC.

4.2 A Detailed Evaluation

The first program we will evaluate is the N-Queens benchmark included in BOTS [7]. Each task in N-Queens spawns 0 to N child tasks, and the depth of its task invocation trees varies from 1 to N, while not following any simple pattern. The size of individual tasks is relatively small.

Figure 7 shows the performance of N-Queens using a variety of compilers and implementations. Four OpenMP versions are shown: GCC, ICC and Insieme with and without task optimization. Additionally, we included the results of a Cilk version and a fully sequential version without any parallel language primitives. The execution time is presented in a log-log plot to improve readability. The

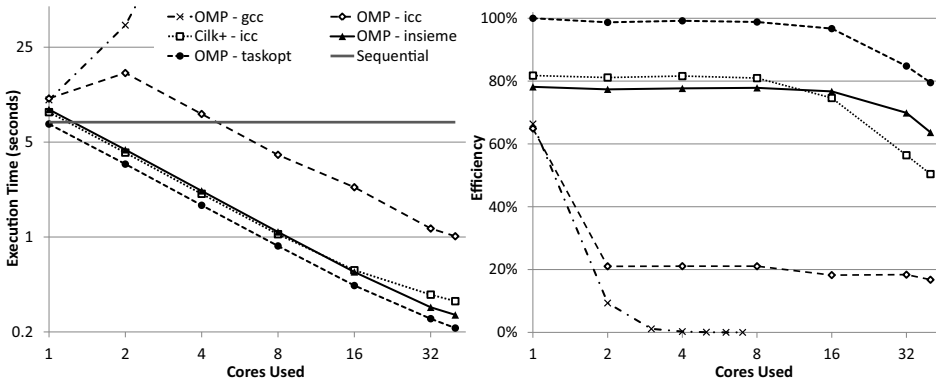


Fig. 7. N-Queens benchmark results, $N = 13$

efficiency plot compares the execution times of the parallel versions against ideal scaling from the sequential version.

In terms of OpenMP results, it is clear that the task granularity in this benchmark is too small to be handled effectively by GCC's GOMP implementation. ICC shows the same behavior that was already partially observed in Section 2 – execution time increases when going from a single-threaded to a multi-threaded setup. However, starting from two threads performance scales relatively well up to 40. Since both of these OpenMP implementations seem ill-equipped to handle very fine-grained tasking well, we also included a Cilk version, which has previously been shown to provide better scaling for fine-grained tasks [15]. Indeed, this implementation performs better in the single-threaded case and scales more smoothly to multiple cores than the GCC and ICC OpenMP versions.

Using Insieme to compile the OpenMP input program results in performance that is comparable to Cilk for up to 16 cores, and scales slightly better beyond this amount. However, a comparison with the fully sequential version indicates that even the Insieme OpenMP version and the Cilk version lose around 20% of performance to overheads incurred due to parallelization. When our task optimization is activated, this overhead is effectively avoided. Even more importantly, this significant reduction in overhead is achieved without negatively affecting the scalability of the program. Performance compared to our implementation without task optimization is improved by 22% to 28% across all measured core counts, with a 25% increase at the full 40 cores.

Compared to the fully sequential version, our approach achieves an efficiency above 99% up to 8 cores, 97% at 16 cores, 85% with 32 cores and 80% at 40 cores. Using the full system (40 cores), our implementation with task optimization improves N-Queens performance by 56% compared to the best competing implementation (Cilk).

4.3 Further Benchmarks

Table 2 summarizes our benchmark results. It includes measurements for the N-Queens benchmark presented above, as well as a number of additional programs.

Sort. Is the *sort* benchmark included in BOTS.

Strassen. Also from BOTS, matrix multiplication using the Strassen algorithm.

Fib. The BOTS *fibonacci* benchmark, remarkable for its very small task size.

Stencil. A task based 2D stencil computation using the cache-oblivious algorithm presented by Frigo and Strumpen [10]. We included this benchmark to represent an important category of cache-oblivious divide-and-conquer algorithms.

Floorplan. The BOTS *floorplan* benchmark. For this application, the binary generated by ICC 12.1 repeatably caused a segmentation fault within ICC's OpenMP library, regardless of the number of threads used. Therefore we are unable to present ICC results for this benchmark.

FFT. A parallel fast fourier transform included in BOTS.

QAP. A branch and bound solver for quadratic assignment problems.

Table 2. Benchmark Results

cores	1	2	5	10	20	40	cores	1	2	5	10	20	40
Queens, $N = 13$ - seq: 7.42							**Fib,** $N = 48$ - seq: 31.09						
gcc	10.23	36.29	148.28	308.16	545.22	725.98	gcc	1960.35	17093.63	>15000	>15000	>15000	>15000
icc	10.49	16.04	6.45	3.81	1.60	0.91	icc	1379.84	2705.65	1135.29	569.15	286.41	157.70
ins	8.69	4.35	1.74	0.87	0.46	0.27	ins	742.40	456.95	247.91	196.59	169.50	155.29
opt	6.79	3.41	1.48	0.69	0.36	0.21	opt	27.06	13.77	6.37	3.30	1.93	1.03
imp	27.92%	27.52%	17.78%	26.64%	25.35%	24.91%	imp	26.43×	32.17×	37.90×	58.15×	86.69×	150.36×
Sort, $N = 2^{27}$ - seq: 21.51							**Strassen,** $N = 8192$ - seq: 158.15						
gcc	21.98	11.80	7.20	17.17	29.43	42.29	gcc	159.74	92.45	39.20	22.10	15.36	19.94
icc	23.87	12.36	5.04	2.80	1.85	1.56	icc	164.43	89.94	39.12	21.81	15.69	19.27
ins	22.94	12.00	4.90	2.71	1.93	1.53	ins	168.84	85.97	37.51	21.98	12.94	8.72
opt	20.81	11.18	4.61	2.52	1.72	1.41	opt	154.27	79.80	35.46	19.81	12.03	8.11
imp	5.61%	5.47%	6.43%	7.47%	7.88%	8.11%	imp	3.54%	7.72%	5.77%	10.08%	7.55%	7.52%
Stencil, $N = 2048$ - seq: 18.90							**Floorplan,** `input.20` - seq: 17.86						
gcc	46.82	62.09	138.51	398.05	576.83	840.61	gcc	27.36	31.04	133.30	352.94	514.51	759.20
icc	30.17	24.65	15.63	14.64	13.84	12.04	icc	*	*	*	*	*	*
ins	32.49	18.48	9.27	6.31	7.50	9.67	ins	23.53	12.48	5.05	2.53	1.72	1.58
opt	24.96	13.84	6.66	4.26	5.15	7.54	opt	17.20	9.51	4.12	2.09	1.43	1.24
imp	20.87%	33.49%	39.17%	47.97%	45.50%	28.29%	imp	36.76%	31.25%	22.62%	21.06%	20.52%	27.68%
FFT, $N = 2^{29}$ - seq: 184.78							**QAP,** `chr18a` - seq: 237.28						
gcc	222.27	132.66	95.88	276.81	420.00	482.07	gcc	488.97	931.43	7471.11	>15000	>15000	>15000
icc	189.73	112.13	55.95	37.44	22.64	16.03	icc	785.36	2539.80	823.00	319.87	179.58	114.93
ins	187.36	104.85	51.39	36.46	21.01	16.96	ins	578.57	294.13	112.80	78.65	70.97	60.71
opt	183.97	100.02	49.66	35.08	19.07	12.03	opt	231.62	110.76	40.24	21.88	15.18	9.90
imp	1.84%	4.84%	3.48%	3.93%	10.16%	33.21%	imp	2.11×	2.66×	2.80×	3.59×	4.68×	6.13×

For every benchmark, the table contains five rows. The results achieved using the GCC and ICC OpenMP implementations are listed in the "gcc" and "icc" rows, respectively. The "ins" row contains the results of our Insieme compiler and runtime without the task multiversioning optimization presented in this paper, while it is enabled for the measurements listed in the "opt" row. Finally, the values in the "imp" row represent the relative improvement achieved using adaptive granularity control, compared to the best result among the other three versions. The columns labeled 1 to 40 correspond to the number of cores used for the computation. All times are given in seconds, and the improvement is provided in percent, except in the case of the Fibonacci and QAP benchmarks where improvement factors are listed instead of very large percentages.

As a frame of reference, the purely sequential time for each benchmark compiled with ICC is provided in each header ("seq"). Note that this time falls between the Insieme time without optimization and the optimized version in most cases, except in the stencil test. Here, the restructuring performed by our compiler prevents some of the low-level sequential optimizations performed by ICC. However, our optimized version executed with one thread is still closer to the sequential performance than any other implementation.

A general trend visible throughout all the benchmark results is the relationship between default task granularity, scaling in GCC and the degree of improvement possible using adaptive task multiversioning and selection. The fibonacci and QAP benchmarks have the most fine grained tasks, and consequently the worst scaling in GCC and the largest improvement with our optimization. On the other end of the spectrum, the FFT, strassen and sort benchmarks feature built-in cutoff values that inherently control task granularity by preventing very small tasks from being generated, resulting in more modest, but still

significant, performance improvements with multiversioning. Floorplan, stencil and N-queens fall in between these extremes.

One interesting behavioral pattern which merits some explanation occurs in FFT. Our multiversioning implementation does not result in any significant improvement up to 10 cores, however at 40 cores the measured improvement is 33%. This is due to the FFT benchmark consisting of two separate phases: coefficient calculation and FFT computation. These phases exhibit distinct scaling behaviour, and one of them is affected more significantly by adaptive granularity optimization than the other. Thus, with a larger number of cores, the phase with bad scaling starts to take up a larger portion of the execution time, and the effect of multiversioning on overall performance increases.

5 Related Work

Much previous work on parallel tasks has focused on runtime systems [3] or scheduling policies [16]. As described in section 2, pure runtime modifications are incapable of dealing with all the causes for inefficiency that our combined compiler and runtime approach covers. Moreover, our proposed multiversioning scheme is orthogonal to scheduling decisions and can be combined with any scheduling policy.

A common approach towards dealing with task granularity issues is having the user provide thresholds or cut-off values [8]. In our work, task granularity is controlled entirely by the compiler and runtime system, without requiring manual programmer support. Duran et al. [6] describe an adaptive cut-off method which does not require manual adjustment, but their pure runtime approach does not offer the performance benefit of full sequentialization in the compiler.

Inlining of recursive functions has been previously performed in sequential program transformation [9], even with the express purpose of improving performance in divide and conquer programs by reducing overheads [18]. However, these works do not deal with parallelism, while our approach focuses primarily on minimizing the overhead incurred by parallel task creation and synchronization.

Some recent publications have used compiler multiversioning in a parallel setting [4][13], but they focused exclusively on loop-based data parallelism. Conversely, our multiversioning approach is designed for task-parallel, recursive programs.

Very recently, Deshpande and Edwards used recursion unrolling to improve opportunities for parallelism in Haskell programs [5]. Unlike our method, they do not use multiversioning or version selection at runtime, and their compiler transformations are designed for the Haskell functional language while we process input programs written in C with OpenMP.

6 Conclusion

We have presented a fully automatic, adaptive approach to parallel task granularity control which goes beyond what can be achieved by improving either

just a runtime system or focusing only on compilation. By combining a compiler which performs task multiversioning with a runtime system that adaptively selects from these versions, we were able to minimize parallel runtime overhead even for very fine grained tasks. Our method uses a novel combination of compiler transformations to build an optimized set of semantically equivalent task versions which differ in granularity. The availability of this set of implementations in the compiled program in turn enables our runtime heuristic to adjust the amount of tasks generated, while incurring even less overhead than a traditional lazy task creation system with cut-offs.

Evaluating our proposed method across a set of eight benchmarks has shown that our optimization is widely applicable, and that the magnitude of these improvements is related to the task granularity of the input program. For programs with relatively coarse-grained tasks, execution times are reduced by 5% - 10%, while we can achieve improvements of a factor of 6 or more compared to the best competing implementations in fine-grained test cases. Benchmark results also demonstrate that our runtime selection heuristic successfully ensures that scalability (up to 40 cores) is not negatively affected by adaptive task granularity adjustment. Crucially, our adaptive granularity control scheme improves performance in all tested benchmarks and for any given number of cores.

Acknowledgments. This research was partially funded by the Austrian Research Promotion Agency under contract nr. 834307 (AutoCore) and the chistera project GEMSCLAIM. We would also like to thank the reviewers for their insightful comments.

References

[1] Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. rep. EECS Department, University of California (2006)
[2] Blumofe, R.D., et al.: Cilk: an efficient multithreaded runtime system. In: Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP 1995, Santa Barbara, California, United States, pp. 207–216 (1995)
[3] Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an efficient openMP runtime system for both fork-join and data flow paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)
[4] Chen, X., Long, S.: Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning. In: Proc. 15th Int. Conf. on Parallel and Distributed Systems, ICPADS 2009, pp. 907–912 (2009)
[5] Deshpande, N.A., Edwards, S.A.: Statically Unrolling Recursion to Improve Opportunities for Parallelism. Tech. rep. Department of Computer Science, Columbia University (2012)
[6] Duran, A., et al.: An adaptive cut-off for task parallelism. In: Proc. 2008 ACM/IEEE Conf. on Supercomputing, SC 2008, Austin, Texas, pp. 36:1–36:11 (2008)
[7] Duran, A., et al.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Proc. 2009 Int. Conf. on Parallel Processing, ICPP 2009, pp. 124–131 (2009)

[8] Turner, D.N. (ed.), et al.: On the Granularity of Divide-and-Conquer Parallelism. In: Glasgow Workshop on Functional Programming, pp. 8–10. Springer (1995)
[9] Fitzpatrick, S., et al.: Unfolding Recursive Function Definitions Using the Paradoxical Combinator (1996)
[10] Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proc. 19th Int. Conf. on Supercomputing, ICS 2005, Cambridge, Massachusetts, pp. 361–366 (2005)
[11] Insieme Compiler and Runtime Infrastructure. Distributed and Parallel Systems Group, University of Innsbruck, `http://insieme-compiler.org`
[12] Intel. Intel C and C++ Compilers (2012), `http://software.intel.com/en-us/c-compilers/`
[13] Jordan, H., et al.: A Multi-Objective Auto-Tuning Framework for Parallel Codes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Article No. 10. IEEE Computer Society Press, Los Alamitos (2012)
[14] Mohr, E., et al.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems 2 (1991)
[15] Olivier, S., Prins, J.F.: Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming 38(5-6), 341–360 (2010)
[16] Olivier, S.L., et al.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. 26(2), 110–124 (2012)
[17] OpenMP Architecture Review Board. OpenMP Specification. Version 3.1 (2011), `http://www.openmp.org/mp-documents`
[18] Rugina, R., Rinard, M.: Recursion Unrolling for Divide and Conquer Programs. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 34–48. Springer, Heidelberg (2001)
[19] Stallman, R.: Using and Porting the GNU Compiler Collection. In: M.I.T. Artificial Intelligence Laboratory (2001)