Extended Overhead Analysis for OpenMP

Michael K. Bane and Graham D. Riley

Centre for Novel Computing,
Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester, UK
{bane, griley}@cs.man.ac.uk

Abstract. In this paper we extend current models of overhead analysis to include complex OpenMP structures, leading to clearer and more appropriate definitions.

1 Introduction

Overhead analysis is a methodology used to compare achieved parallel performance to the ideal parallel performance of a reference (usually sequential) code. It can be considered as an extended view of Amdahl's Law [1]:

$$T_p = \frac{T_s}{p} + \frac{p-1}{p}(1-\alpha)\,T_s \tag{1}$$

where T_s and T_p are the times spent by a serial and parallel implementation of a given algorithm on p threads, and α is a measure of the fraction of parallelized code. The first term is the time for an ideal parallel implementation. The second term can be considered as an overhead due to unparallelized code, degrading the performance. However, other factors affect performance, such as the implementation of the parallel code and the effect of different data access patterns. We therefore consider (1) to be a specific form of

$$T_p = \frac{T_s}{p} + \sum_i O_i \tag{2}$$

where each O_i is an overhead. Much work has been done on the classification and practical use of overheads of parallel programs eg ([2], [3], [4], [5]).

A hierarchical breakdown of temporal overheads is given in [3]. The top level overheads are information movement, critical path, parallelism management, and additional computation. The critical path overheads are due to imperfect parallelization. Typical components will be load imbalance, replicated work and insufficient parallelism such as unparallelized or partially parallelized code. We extend the breakdown of overheads with an "unidentified overheads" category that includes those overheads that have not yet been, or cannot be, determined during the analysis of a particular experiment. It is possible for an overhead

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 162–166.

to be negative and thus relate to an improvement in the parallel performance. For example, for a parallel implementation the data may fit into a processor's memory cache whereas it does not for the serial implementation. In such a case, the overhead due to data accesses would be negative.

The practical process of quantifying overheads is typically a refinement process. The main point is not to obtain high accuracy for all categories of overheads, but to optimize the parallel implementation.

Overhead analysis may be applied to the whole code or to a particular region of interest.

2 Overhead Analysis Applied to OpenMP

This paper argues that the current formalization of overhead analysis as applied to OpenMP [6] is overly simplistic, and suggests an improved scheme.

Consider two simple examples to illustrate the definition and measurement of an overhead. A simple `OMP PARALLEL DO` loop may lead to load imbalance overhead, defined as the difference between the time taken by the slowest thread and the average thread time. The definition of the Load Imbalance overhead in [3] is given as

Load imbalance: time spent waiting at a synchronization point, because, although there are sufficient parallel tasks, they are asymmetric in the time taken to execute them.

We now turn our attention to the simplest case of unparallelized code overhead, where only one thread executes code in a given parallel region – for example, an `OMP PARALLEL` construct consisting solely of an `OMP SINGLE` construct. From [3] we have the following definitions:

Insufficient parallelism: processors are idle because there is an insufficient number of parallel tasks available for execution at that stage of the program; with subdivisions:

Unparallelized code: time spent waiting in sections of code where there is a single task, run on a single processor;

Partially parallelized code: time spent waiting in sections of code where there is more than one task, but not enough to keep all processors active.

For the above examples we have a synchronization point at the start and end of the region of interest, and only one construct within the region of interest. However, analysis of such examples is of limited use. OpenMP allows the creation of a parallel region in which there can be a variety of OpenMP constructs as well as replicated code that is executed by the team of threads [1]. The number of threads executing may also depend on program flow; in particular when control is determined by reference to the value of the function `OMP_GET_THREAD_NUM`,

[1] OpenMP allows for differing numbers of threads for different parallel regions, either determined by the system or explicitly by the user. In this paper, we assume that there are p threads running for each and every parallel region. Cases where there a different number of threads for a parallel region is beyond the scope of this introductory paper.

which returns the thread number. Various OpenMP constructs can also *not* have an implicit barrier at the exit point (for example, `OMP END DO NOWAIT`). Thus a given OpenMP parallel region can be quite sophisticated leading to several different overheads within a region which may interfere constructively or destructively. The remainder of this paper discusses appropriate overhead analysis for non-trivial OpenMP programs.

Let us now consider an OpenMP parallel region consisting of a `SINGLE` region followed by a distributed `DO` loop:

```
C$OMP PARALLEL PRIVATE(I)
C$OMP SINGLE
      CALL SINGLE_WORK()
C$OMP END SINGLE NOWAIT
C$OMP DO SCHEDULE(DYNAMIC)
      DO I=1, N
         CALL DO_WORK()
      END DO
C$OMP END DO
C$OMP END PARALLEL
```

Since the `SINGLE` region does not have a barrier at the exit point, those threads not executing `SINGLE_WORK()` will start `DO_WORK()` immediately. We could therefore have a situation shown in Figure 1, where the double line represents the time spent in `SINGLE_WORK()`, the single line the time spent in `DO_WORK` and the dashed line being thread idle time. One interpretation of the above

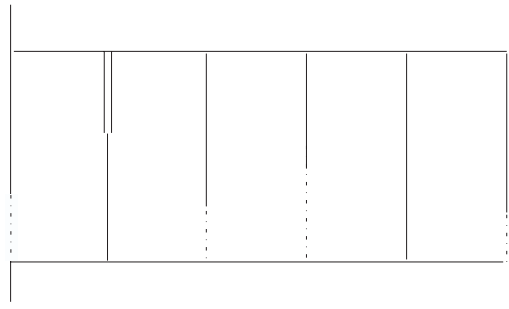

Fig. 1. Time Graph for Complex Example #1

definitions would be that this example has an element of unparallelized code overhead.

Depending upon the amount of time it takes to perform `SINGLE_WORK()` it is possible to achieve ideal speed up for such an example, despite a proportion of code being executed on only one thread, which would normally imply unparallelized code overhead.

Assume the time spent on one thread is t_{sing} for `SINGLE_WORK()`and t_{do} for `DO_WORK()` then for this region the sequential time $T_s = t_{sing} + t_{do}$ and the ideal time on p threads is thus $T_{ideal} = \frac{T_s}{p} = \frac{t_{sing}+t_{do}}{p}$. During the time that one thread has spent in the `SINGLE` region a total of $(p-1)\,t_{sing}$ seconds have been allocated to `DO_WORK()`. There is therefore $t_{do} - (p-1)\,t_{sing}$ seconds worth of work left to do, now over p threads. So, the actual time taken is

$$T_p = t_{sing} + \max\left(0, \frac{t_{do} - (p-1)\,t_{sing}}{p}\right) \tag{3}$$

Thus either the work in the `SINGLE` region dominates (all the other threads finish first), or there is sufficient work for those threads executing `DO_WORK()` compared to `SINGLE_WORK()` in which case (3) reduces to $T_p = T_{ideal}$. That is, we may achieve a perfect parallel implementation despite the presence of a `SINGLE` region; perfection is not guaranteed, depending on the size of the work quanta in `DO_WORK`.

Therefore, we can see that the determination of overheads needs to take into account interactions between OpenMP constructs in the region in question.

Consider a slight variation to the above case, where an OpenMP parallel region contains just an `OMP SINGLE` construct and an `OMP DO` loop without an exit barrier (ie `OMP END DO NOWAIT` is present). As long as the work is independent, we can write such a code in two different orders, one with the `SINGLE` construct followed by the `DO` loop and the other in the opposite order.

At first glance, one might be tempted to define the overheads in terms of that OpenMP construct which leads to lost cycles immediately before the final synchronization point. Thus overhead in the first case would be mainly load imbalance with an unparallelized overhead contribution, and in the second case, mainly unparallelized overhead with a load imbalance overhead contribution.

Given such "commutability" of overheads, together with the previous examples, it is obvious we need a clearer definition of overheads.

3 An Improved Schema

We now give a new, extended schema for defining overheads for real life OpenMP programs where we assume that the run time environment allocates the requested number of threads, p, for each and every parallel region.

1. Overheads can be defined only between two synchronization points. Overheads for a larger region will be the sum of overheads between each consecutive pair of synchronization points in that region.
2. Overheads exist only if the time taken between two synchronization points by the parallel implementation on p threads, T_p, is greater than the ideal time, T_{ideal}.
3. Unparallelized overhead is the time spent between two consecutive synchronization points of the parallel implementation when only one thread is executing.

4. Partially parallelized overhead is the time spent between two synchronization points when the number of threads being used throughout this region, p', is given by $1 < p' < p$. This would occur, for example, in an `OMP PARALLEL SECTIONS` construct where there are less `SECTION`s than threads.
5. Replicated work overhead occurs between two synchronization points when members of the thread team are executing the same instructions on the same data in the same order.
6. Load imbalance overhead is the time spent waiting at the exit synchronization point when the same number of threads, $p'' > 1$, execute code between the synchronization points, irrespective of the cause(s) of the imbalance. In the case $p'' < p$, we can compute load imbalance overhead with respect to p'' threads and partially parallelized overhead with respect to $p - p''$ threads.

In computing overheads for a synchronization region, point (2) should be considered first. That is, if there is ideal speed up, there is no need to compute other overheads – ideal speed up being the "goal". There may, of course, by some negative overheads which balance the positive overheads but this situation is tolerated because the speed up is acceptable.

4 Conclusions and Future Work

In this paper we have outlined an extension to the current analysis of overheads, as applied to OpenMP. Our future work will involve expanding the prototype Ovaltine [5] tool to include these extensions, and an in-depth consideration of cases where different parallel regions have different numbers of threads, either as a result of dynamic scheduling or at the request of the programmer.

References

1. G.M. Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, AFIPS Conference Proceedings, vol. 30, AFIPS Press, pp. 483-485, 1967.
2. M.E. Crovella and T.J. LeBlanc, Parallel Performance Prediction Using Lost Cycles Analysis, Proceedings of Supercomputing '94, IEEE Computer Society, pp. 600-609, November 1994.
3. J.M. Bull. *A Hierarchical Classification of Overheads in Parallel Programs*, Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems, I. Jelly, I. Gorton and P. Croll (Ed.s), Chapman Hall, pp. 208-219, March 1996.
4. G.D. Riley, J.M. Bull and J.R. Gurd, *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics*, Proc. 11^{th} ACM International Conference on Supercomputing, ACM Press, pp. 36-43, July 1997.
5. M.K. Bane and G.D. Riley, *Automatic Overheads Profiler for OpenMP Codes*, Proceedings of the Second European Workshop on OpenMP (EWOMP2000), September 2000.
6. http://www.openmp.org/specs/