Automatic Overheads Profiler for OpenMP Codes

M.K. Bane and G.D. Riley,
Centre for Novel Computing, University of Manchester, M13 9PL, UK.
email: [mkbane, griley]@cs.man.ac.uk

Abstract

To develop a good parallel implementation requires understanding of where run-time is spent and comparing this to some realistic best possible time. We introduce "overhead analysis" as a way of comparing achieved performance with achievable performance. We present a tool, OVALTINE, which aims to provide, automatically, the user with a hierarchical set of overheads for a given OpenMP implementation with respect to a given serial implementation. We give preliminary results from OVALTINE on an SGI Origin2000, and show how our tool can be used to improve performance.

Introduction

Developing "good" parallel code for shared memory architectures requires both portability and efficient execution of the code. OpenMP addresses the portability issue but, for any given program, there are many possible parallel implementations with varying efficiencies. When developing a parallel code a user will typically make some modifications then profile the modified code. Depending upon these new timings, the user may discard the modifications or incorporate them. Usually, the user will make many passes through this "modify–measure" loop.

Often, and naively, when users profile their code they only look at the actual time taken, concentrating on the time consuming sections. This draws them into focussing, during each pass of the "modify–measure" loop, on a single section of code. The user may also be working with no clear indication of the maximum possible performance. Both these situations may lead to wasted effort. We introduce "overhead analysis" as a method to help prevent this.

What is Overhead Analysis and How is it Useful?

Overhead analysis is a technique to provide developers with more information about the execution of their code, specifically to help determine the maximum performance possible. Overhead analysis is an extended view of Amdahl's Law, as we now explain.

Assume T_s is the time spent by a serial implementation of a given algorithm and T_p is the time spent by a parallel implementation of the same algorithm on p threads. Then for perfect parallelisation we would have $T_p = T_s/p$.

Amdahl's Law introduces α as a measure of the fraction of parallelised code in the parallel implementation, and states that:

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

Thus, the best time for a parallel implementation is restricted by the fraction $(1-\alpha)$ of unparallelised code. Let us rearrange equation (1) to give

$$T_p = \frac{T_s}{p} + \frac{p-1}{p}(1-\alpha)T_s \qquad (2)$$

The first term is the time for an ideal parallel implementation. The second term can be considered as an overhead or degradation of the performance. In this case it is an overhead due to unparallelised code. However, this model is too simplistic in that it takes no account of any of the various factors affecting performance, such as how well the parallelised code has been implemented. Let us therefore consider equation (2) to be a specific form of

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

where each O_i is an overhead.

A hierarchical classification of overheads for parallel programs has been given by Bull [1]. In

this paper we are only interested in temporal overheads – how does the measured parallel time differ from the ideal parallel time. Work on spatial overheads – how do the memory requirements of the parallel implementation differ from those of the sequential implementation – will be considered in an extension of the current work. The use of overhead analysis to improve the performance of a molecular dynamics code is illustrated by Riley et al [2].

We use a hierarchical breakdown of temporal overheads based on [1] and [2], see Table 1. The top level overheads being Information Movement (e.g. data access costs), Critical Path, Parallelism Management (e.g. costs of any additional code to manage the parallelism), and Additional Computation (e.g. changes to expose parallelism). The Critical Path overheads are due to imperfect parallelisation. Typical components will be load imbalance (some threads taking longer to complete their work than others), replicated work (it may be cheaper to replicate work than force a synchronisation) and non-parallelised, or partially-parallelised, code.

We also introduce the "Unidentified Overheads" category which includes those overheads that have not yet been, or cannot be, determined.

Note that it is possible for an overhead to be negative and thus relate to an *improvement* in the parallel performance. For example, for a certain number of processors it may be possible that the data fits into cache when it does not for the serial implementation. In such a case, the overhead due to data accesses would be negative. This may lead to an overall super linear speed up for the parallel implementation.

Note that the process of quantifying overheads is typically a refinement process. Initially, just a few overheads are determined. If this leads to a large figure in the Unidentified Overheads category, then further timing experiments may be needed to determine other overheads. Alternatively, further analysis of the existing performance data might reveal the composition of the unidentified overheads. Furthermore, it may be that overhead analysis is of interest only for a particularly time consuming area of code, rather than the complete program.

Overheads, OVALTINE and OpenMP

OVALTINE is a tool which, where possible, automates the determination of the overheads of a given OpenMP implementation with respect to a given serial implementation. OVALTINE can give the overheads either for the complete program or for a specific region, down to a basic block or DO loop, for example.

OVALTINE takes two source codes: a serial and an OpenMP implementation of the same basic algorithm. Currently, OVALTINE only supports Fortran 77 source files with (fixed form) OpenMP directives but it is intended to extend the tool to support Fortran 90 and C also. OVALTINE instruments each input file appropriately, manages execution of the instrumented codes (using any makefile or other specific instructions) on a user-specified set of numbers of threads, and analyses the data to produce a table of overheads. The user is then able to determine whether and where to invest effort in further code parallelisation.

By using overhead analysis users are able to determine whether they have achieved the maximum realistic performance (without algorithmic changes). A tool such as OVALTINE which automates this approach will further help in the rapid development of efficient OpenMP programs.

OVALTINE: Outline Design

OVALTINE is a tool designed to help a user determine the values of the relevant overheads for their parallel implementation. It is built in Java and has two main kernels – *instrumentation* and *analysis*. Figure 1 gives a schematic outline of OVALTINE. Travelling down the diagram is the instrumentation stage. This occurs for both the serial and parallel implementations. Once instrumented, these codes are run giving the data (sets of timings and counts). Travelling from right to left, to obtain the overheads from the data, is the analysis stage. Note that the "which experiment?" information is used both for the instrumentation stage (given the code, what do we need to count and time in order to determine the overheads) and the analysis stage (given the times and counts for this section of code what experiment did we perform and thus which overheads are we computing). Future developments will include making the "which experiment?" section into a separate, extensible, expert knowledge based module.

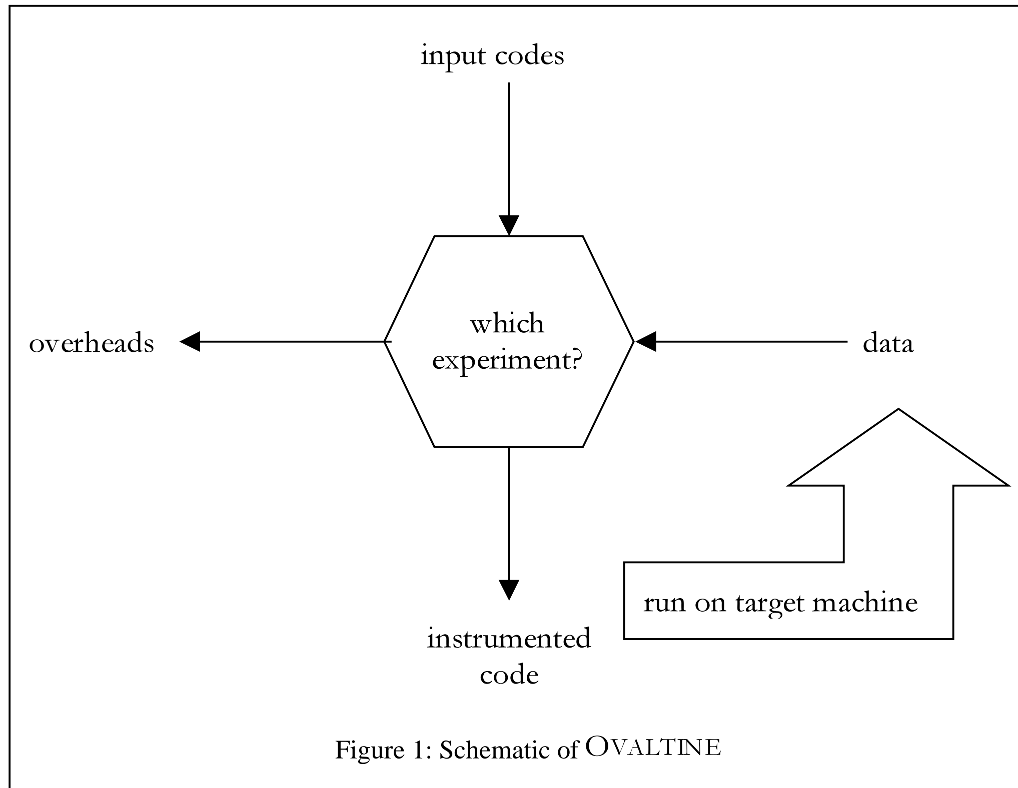

Figure 1: Schematic of OVALTINE

In order to simplify the manipulation of the code during instrumentation and analysis, the code is parsed and a tree of nodes created. Each node is numbered for use in the analysis part of OVALTINE. Nominally, nodes will be either OV nodes (see below), OMP nodes or nodes relating to the Fortran source. The latter will include:

- "root node" – the node containing the name of the current subprogram unit
- "Declarations" – type declarations (including COMMON) and no executable statements
- "BB" – basic blocks are sections of code that contain no control structure (assignments, I/O, comments, blank lines)
- "DO" or "ENDDO" (the latter includes any ending for a DO loop)
- "IF", "ELSE" or "ENDIF"
- "CALL" – contains name of subroutine being called (plus actual arguments). This node will also contain a reference to a new tree which contains the code relating to that subroutine. This saves replication (if a subroutine is called more than once) and gives greater flexability in handling routines not contained in the input source file or even those whose source is unavailable (e.g. calls to a library).
- "STOP" and "END" nodes

Currently we use Polaris scanner [3] to take a Fortran 77 file and build a basic abstract syntax tree. (Polaris is limited to Fortran 77 input files.)

The Polaris scanner will pick up OpenMP directives. (One limitation of Polaris is that it only recognises directives of the form "C$OMP".) We can thus parse the parallel input file and include suitable OMP nodes in our tree. The current OVALTINE implementation has a different OMP type node for each possible OpenMP directive.

For each of the serial and parallel implementations, OVALTINE requires a valid output file from the Polaris scanner and the original Fortran source. OVALTINE uses these to produce an internal representation which is a tree of nodes of the types described above.

As an example, consider the segment of code:

```
      PROGRAM example
      IMPLICIT none
      INTEGER i,n
      REAL a(10), b(10), c(10)
      READ(5,*) b,c,n
      IF (n.le.10) THEN
C$OMP    PARALLEL DO PRIVATE(i)
         DO i=1,n
            a(i) = b(i) + c(i)
         END DO
C$OMP    END PARALLEL DO
      ELSE
        WRITE(6,*) 'n is too big'
      END IF
      STOP
      END
```

The internal tree for this code segment is given in Figure 2.

OVALTINE: Instrumention Stage

Once we have a tree of nodes for both the serial and parallel input files, we can instrument them accordingly. In order to determine overheads we use a mixture of counters and timers. Counters determine how many times particular constructs are executed at run-time and timers help determine how long various operations take. In order to compute some overheads we need to record the start and stop times for various OpenMP constructs and code segments on each iteration and for each thread. We use an array to store the "timer on" times, such as:

```
timer_on(thread, program
unit, node, iteration)
```

Similarly for "timer off" times.

Ideally we would know the maximum values of the array indices during the instrumentation phase of OVALTINE so that we can keep memory use to a minimum. Rather than use the node numbering to distinguish the time being recorded, it is possible to refer to the number of the node being monitored to reduce the size of the third dimension to the number of nodes being monitored (rather than the number of the highest node being monitored).

Note that the fourth array index would only be used when the overhead being measured is within a loop. For example, if we have two DO loops and the inner most one is the parallel one, then we can determine the load imbalance per iteration.

To minimise intrusion on the original code we only add timers or counters to those code regions where we wish to determine an overhead, rather than time and count every basic block and DO loop. It remains an open research question how best to manage such intrusion.

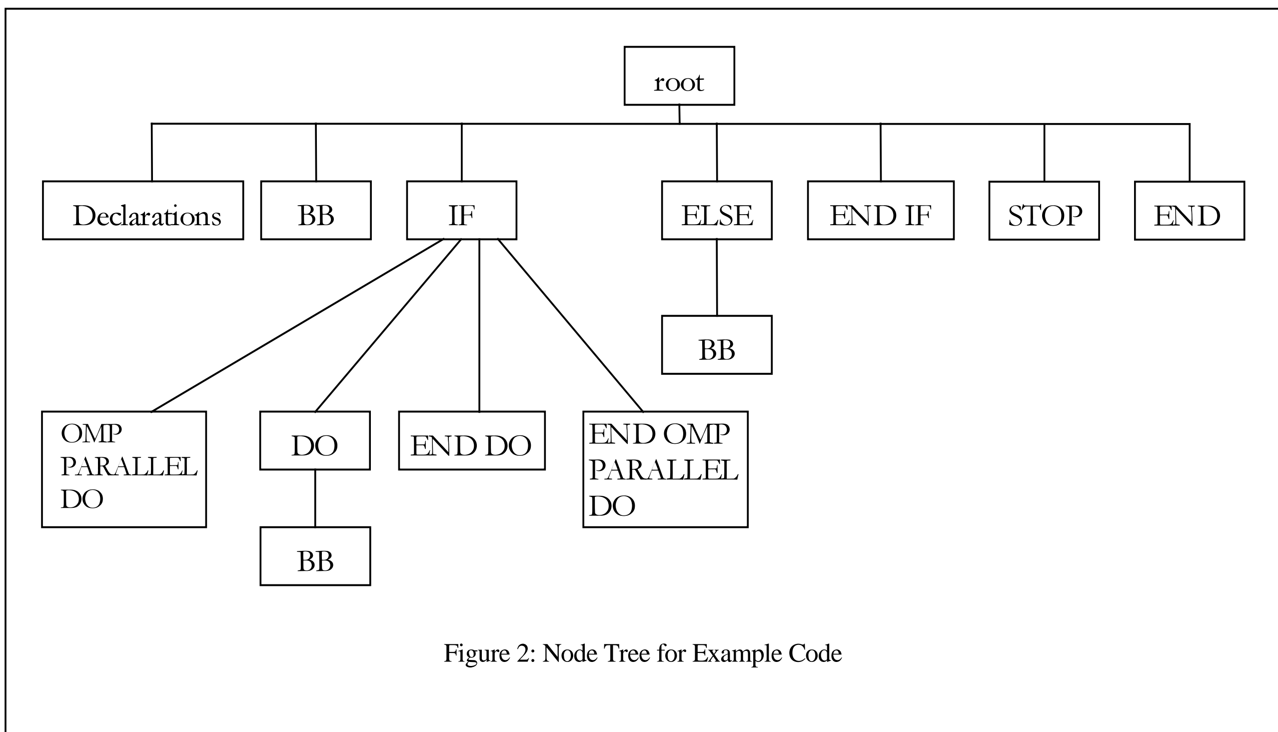

Figure 2: Node Tree for Example Code

Measuring the Overheads

Table 1: Overheads and their Measurements

Overhead Category	***Sub-categories***		***Measurement Method***
Information Movement	Data accesses		(under investigation)
	Synchronisation	locks	empirically determine the time to take and release a lock, then count the number of times a lock is encountered. Ongoing research into how best to measure the lock contention time
		barriers	empirically determine the cost of a barrier, then multiply by the count of the number of times a barrier is encountered.
Critical Path	Load imbalance		time each thread and determine difference between maximum time and average time
	Replicated work		insert timers around replicated code
	Insufficient parallelism	unparallelised code	upper bound given by inserting timers to accumulate time outside of `OMP PARALLEL` regions
		partially parallelised code	(under investigation)
Management of Parallelism	User scheduling		empirical computation (e.g. of different `SCHEDULE` clauses) and multiply by run time count
	Run time system scheduling		(under investigation)
Additional Computation	Implementation changes		comparison of sequential and parallel Fortran source
	Instruction level changes		comparison of sequential and parallel assembler source
Unidentified	Overheads not covered in the above categories		difference between measured and ideal parallel times not included in the other overheads

Table 1 gives the overheads that OVALTINE computes, and indicates how these are measured. As can be seen, it is relatively easy to identify the overheads categories. However, practical measurement may be harder and work is on-going to determine the best way to measure some overheads.

Generally, overheads are computed by instrumenting the original source code with counters (to count certain events) and timers (to time certain events). Such additions to the original code are made by the insertion of additional nodes to the internal tree. We label these as OV nodes for ease in the analysis stage.

Apart from instrumenting all the codes to determine the required overheads, we also need a means of handling the data collected. For this purpose, OVALTINE adds a call to subprogram `OV_output` which is compiled together with each of the instrumented files. This routine essentially outputs data in a standard layout that the analysis part of OVALTINE can read.

Timers and counters are used to ensure portability across platforms. We are currently investigating the appropriateness of using the PAPI [4] standards initiative to allow OVALTINE to make use of hardware specific performance analysis tools.

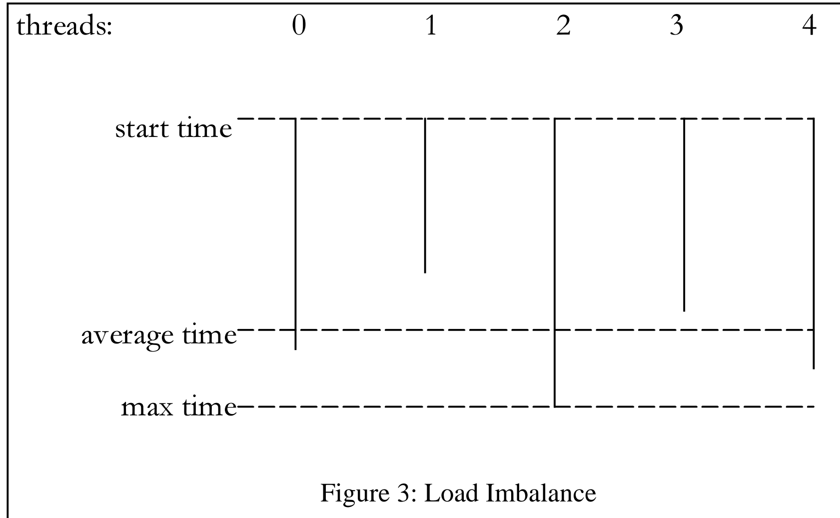

Figure 3: Load Imbalance

We illustrate our general methodology by outlining how to determine the load imbalance overhead. For an `OMP DO` loop that is spread over several threads we note that an unbalanced load will result in different threads taking differing amounts of time to complete their iterations. For example, we could have the situation as shown in Figure 3.

As can be seen, this is poorly load balanced. Thread 2 is taking much longer than the other threads and thus determines the overall time for this loop. If the load were perfectly balanced, the overall time taken would be the average time. We therefore define the overhead due to load imbalance for a `DO` loop as the difference between the maximum and average thread times.

We need to consider the various types of parallel `DO` loops that we might come across. There is the combination `OMP PARALLEL DO` (with or without explicit `OMP END PARALLEL DO`) or a `OMP DO` construct in an already parallel region. (For the purposes of simplicity we assume no nested parallelism.) Note that the `OMP END PARALLEL DO` construct may or may not be present and, if present, may have a `NOWAIT` clause.

To determine the load imbalance we require each thread to start a timer before entering the `OMP DO` loop and to stop that timer as soon as it has finished the `DO` loop work, *without* synchronising with the other threads. If t_j is the time taken on thread j and there are n threads, then the load imbalance is

$$\max_j(t_j) - \frac{1}{n}\sum_j t_j$$

For example, consider the combination `OMP PARALLEL DO`. Table 2 lists one example of the original code together with the corresponding instrumented code, with differences given in ***bold italics***.

Table 2: Example of Instrumenting an OMP PARALLEL DO Construct

input F77 source code	instrumented code
C$OMP PARALLEL ***DO*** &	C$OMP PARALLEL &
C$OMP *parallel_do_clauses* &	C$OMP *parallel_do_clauses* &
C$OMP SHARED(shared_list)	C$OMP SHARED(shared_list***, timer_on, timer_off***)
	timer_on(omp_get_thread_num(),PU, node, iter)
	C$OMP DO
DO …	DO …
do_loop_body	do_loop_body
END DO	END DO
	C$OMP END DO NOWAIT
	timer_off(omp_get_thread_num(),PU, node, iter)
C$OMP END PARALLEL ***DO***	C$OMP END PARALLEL

OVALTINE: Analysis Stage

Ideally, OVALTINE would handle the compilation and execution of the instrumented codes. Note that it is important to ensure that we know how many threads are actually being used in each parallel region. (The current version assumes the same number of threads for all parallel regions.) One way to guarantee the number of threads would be to set OMP_DYNAMIC to false. Furthermore, in order to obtain sensible figures for the overheads, we need reproducible timings over runs if possible.

The analysis part of OVALTINE reads in the results and uses the input Fortran codes, together with the corresponding Polaris generated abstract syntax tree, to compute the overheads. That is, OVALTINE does not rely on knowledge from the previous instrumentation stage. This enables users to use OVALTINE to instrument codes and then run the codes at their convenience (for example if they need to be run in batch or on a quiet machine) and at a later date take the results and original codes and obtain an overhead analysis.

The analysis essentially works backwards from the instrumentation. Given data which include the node number (as well as timings or counts) we can determine the construct to which it relates and thus the overheads experiment that must have been performed.

Results

We have a prototype OVALTINE which takes the input codes, together with output from Polaris scanner. The overheads that the prototype currently computes are load imbalance and unparallelised code. However, other experiments and analysis for other overheads can be included, given the object orientated nature of the prototype. Furthermore, we have not yet extended the prototype to deal with calls to subroutines. (The CALL node is created, but the new tree of nodes is not.)

We take a simple example – forming the matrix product $A=LU$ where L is a lower diagonal matrix and U an upper diagonal matrix. The parallel implementation is shown below:

```
      program LU2
      implicit none
      integer n, limit
      parameter (n=1100)
      double precision a(n,n), l(n,n), u(n,n)
      integer i,j,k
      integer num_threads
      double precision go, stop, timef, t
      call initial(a,l,u,n)
      go = timef()
C$OMP PARALLEL DO default(none)
C$OMP&private(i,j,k)
C$OMP&shared (a,l,u)
C$OMP&schedule(static)
      do i=1,n
         do j=1,n
            do k=1,i
               a(i,j) = a(i,j) + l(i,k)*u(k,j)
            end do
         end do
      end do
C$OMP END PARALLEL DO
      stop = timef()
      t = stop - go
      write(6,*) "time=" t
      end
```

The serial version is the same as above but with the OpenMP directives removed. The `initial` subroutine sets A to zero and initialises the L and U matrices. There are no OpenMP statements in the `initial` subroutine (by design). From this code we would expect to see an unparallelised code overhead (for the initialisation and output statements) and a significant load imbalance overhead (since the amount of work depends on the outer, `i`, loop counter).

Results given here were obtained on an SGI Origin2000 system, running Irix 6.5, using the MIPSpro compiler version 7.3.1.1m. The SGI `miser` resource manager was used to obtain dedicated processor time. Unfortunately, experience has shown that using miser on a reasonably heavily loaded system still results in significant (more than 10%) variation in times between runs. (This is because miser gives dedicated processors and the requested amount of memory, but makes no guarantee to allocate memory physically close to the processors used.) The use of miser does, however, guarantee that the number of threads in each parallel region is `OMP_NUM_THREADS` exactly. This means we do not have to set `OMP_DYNAMIC` to false which, while guaranteeing the number of threads, on an Origin2000 system turns on gang scheduling resulting in increased wall clock times, particularly if the number of threads exceeds the number of "free" processors.

It is possible to reduce the effects of system scheduling, by using an empty machine for example. However, this paper treats these effects as another overhead (which would initially appear in the Unidentified Overheads category).

The results obtained by OVALTINE are given in Table 3 where the naïve ideal parallel time on p threads is just T_s/p and the efficiency E_p is given by

$$E_p = T_s / pT_p$$

where T_s and T_p are the times taken for the serial and parallel implementations respectively.

We note immediately that the efficiency of this implementation is poor – dropping off to less than 50% on just 8 threads.

The major overhead degrading performance is the load imbalance. Altering the schedule of the parallel `DO` loop from a simple to an interleaved distribution (by giving the `SCHEDULE` clause argument `(STATIC, 1)`) should give a more satisfactory load balance in the loop, and therefore a better overall parallel implementation. Rerunning OVALTINE with this new parallel implementation gives the overheads shown in Table 4. As expected, this is a more efficient parallel implementation.

We note that the Unidentified Overheads is quite large in all cases. Given the small amount of sequential work outside the main loop, we deduce that much of the Unidentified Overheads is due to increased data access costs and the Management of Parallelism overheads. Given that we also observed up to 8 seconds variation between serial runs, it is probable that much of the Unidentified Overheads are attributable to variations in run-time. Further experiments could be carried out to confirm these assumptions and classify some of these Unidentified Overheads. This would enable the user to determine whether more effort was justified to reduce further the overheads.

In both cases we see that the non-parallel overhead increases (slowly) with increasing number of threads, as predicted by the theoretical contribution of

$$\frac{p-1}{p}(1-\alpha)T_s$$

where p is the number of threads, T_s the time for the serial code, and α is the fraction of parallelised code.

Table 3: OVALTINE Results for Unbalanced Loop

Number of threads p	2	3	4	8
Serial time/s T_s	233.829	233.829	233.829	233.829
Parallel time/s T_p	199.162	142.384	115.924	66.835
Efficiency E_p	58.70%	54.74%	50.43%	43.73%
Naïve ideal parallel time/s T_s/p	116.914	77.943	58.457	29.229
Total overhead/s $T_p - T_s/p$	82.248	64.441	57.467	37.606
Unparallelalised OH/s	0.357	0.462	0.526	0.616
Load imbalance OH/s	69.940	60.358	52.826	33.449
Unidentified OH/s	11.951	3.621	4.115	3.541

Table 4: OVALTINE Results for Balanced Loop

Number of threads p	2	3	4	8
Serial time/s T_s	233.829	233.829	233.829	233.829
Parallel time/s T_p	127.586	84.519	64.074	33.566
Efficiency E_p	91.63%	92.22%	91.23%	87.08%
Naïve ideal parallel time/s T_s/p	116.914	77.943	58.457	29.229
Total overhead/s $T_p - T_s/p$	10.672	6.576	5.617	4.337
Unparallelalised OH/s	0.346	0.478	0.522	0.574
Load imbalance OH/s	0.028	0.321	1.403	0.447
Unidentified OH/s	10.298	5.777	3.692	3.316

Further Work

Future work will include extending the OVALTINE prototype to compute more categories of overheads, handling all OpenMP directives, including any appearing with v2.0, and allowing Fortran 90 and C as input source files. These latter issues are constrained by the current use of Polaris to create the abstract syntax tree for the source code. We will evaluate alternatives such as SUIF [5], and investigate whether a suitable Java application to create an abstract syntax tree is available so that the use of OVALTINE does not rely on anything other than a JVM and an OpenMP compliant compiler for the given input source codes.

Other issues that will be addressed include: adding functionality to create a suitable script to run the instrumented codes, handling vendor directives in the source codes, fully implementing the handling of calling subroutines, and taking notice of any diagnostic error messages in the Polaris generated files.

Management of the possible intrusiveness of OVALTINE (see above) needs investigating. It will also be interesting to expand OVALTINE to exploit a platform's own performance analysis tools, such as SpeedShop and Perfex on the Origin2000.

Finally, we see OVALTINE as one component of a larger OpenMP program development environment. Future work will integrate OVALTINE with FINESSE[6], or a similar tool, to provide further support both for the parallelisation process, by suggesting program transformations to expose parallelism, and for version control to support the exploration of possible parallel implementations.

Related Work

The large majority of existing tools for parallel performance analysis are aimed at the message-passing programming paradigm. Restricting attention to tools which support Fortran or C, significant research efforts in this area include Paragraph [7], Pablo [8], AIMS [9], Paradyn [10], PMA [11] and XPVM [12]. Commercial systems include Vampir [13] and MPP-Apprentice [14].

Tools for shared memory systems have received less attention, possibly due to the past lack of a widely accepted standard for the associated programming paradigm, and because of the need for hardware support to monitor the memory system. The advent of OpenMP eases the former situation, while the PAPI [4] standards initiative will ameliorate the latter. A number of systems support visualisation of events occurring in threads libraries. Falcon [15] offers real-time event monitoring of thread activity as part of a larger computational steering package. Parade [16] is a visualisation and animation suite which includes support for both threads and message-passing libraries.

Carnival [17] supports waiting time analysis for both message-passing and shared memory systems. This is the nearest to full overheads profiling that any other tool achieves; however, important overhead categories, such as memory accesses, are excluded, and no reference code is used to give an unbiased basis for comparison.

Commercial systems include ATExpert [18] (a precursor to MPP-Apprentice) for Cray vector SMPs, and the ProDev Workshop suite for SGI machines such as the Origin2000 which is a graphical front-end to tools such as SpeedShop and Perfex, enabling profiling of the program counter and hardware performance counters

Conclusions

We have shown that the use of overhead analysis can help pin point the reasons for poor performance in an OpenMP code. Furthermore, we have shown that a tool such as OVALTINE is invaluable in aiding the user to identify, locate and quantify such overheads.

References

[1] 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 (eds), Chapman Hall, pp208-219, March 1996.

[2] G.D. Riley, J.M. Bull and J.R. Gurd. *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics*, Proc. 11th ACM International Conference on Supercomputing, ACM Press, 36-43, July 1997.

[3] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford and K. Faigin. *Polaris: A New-Generation Parallelizing Compiler for MPP's*. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1993.

[4] J. Dongarra, *Performance Data Standard and API*, available at: http://icl.cs.utk.edu/projects/papi/

[5] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng. *The SUIF Compiler for Scalable Parallel Machines*, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, February, 1995.

[6] N. Mukherjee, G.D. Riley and J.R. Gurd. *FINESSE: A Prototype Feedback-guided Performance Enhancement System*, Proc. 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodes, Greece, pp101-109, January 2000.

[7] I. Glendinning, V.S. Getov, A. Hellberg, R.W. Hockney and D.J. Pritchard, *Performance Visualisation in a Portable Parallel Programming Environment*, Proc. Workshop on Monitoring and Visualization of Parallel Processing Systems, Moravany, CSFR, Oct. 1992.

[8] D.A. Reed, *Experimental Analysis of Parallel Systems: Techniques and Open Problems*, Lect. Notes in Comp. Sci. **794**, 25-51, 1994.

[9] J.C. Yan, Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers, Proc. 27th Hawaii Intl. Conf. on System Sciences II, IEEE Comp. Soc. Press, 625-633, Jan. 1994.

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam and T. Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer **28**(11), 37-46, 1995.

[11] B.J.N. Wiley and A. Endo, *Annai/PMA Performance Monitor and Analyzer*, Proc. 4th Intl. Workshop on Modelling, Analysis and Simulation of Comp. and Telecom. Systems (MASCOTS'96), IEEE Comp. Soc. Press, 186-191, Feb. 1996.

[12] G.A. Geist, J. Kohl and P. Papadopoulos, *Visualization, Debugging and Performance in PVM*, Proc. Visualization and Debugging Workshop, Oct. 1994.

[13] W.E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, available at: http://www.kfa-juelich.de/zam/PTdocs/ vampir/vampir.html

[14] W. Williams, T. Hoel and D. Pase, *The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D*, in: K.M. Decker et al. (eds.), *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, 333-345, 1994.

[15] W. Gu, G. Eisenhauer, K. Schwan, J. Vetter, *Falcon: On-line Monitoring and Steering of Parallel Programs*, available at: http://www.cc.gatech.edu/systems/papers/weiming98.ps

[16] J.T. Stasko, *The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report*, Tech. Rep. GIT-GVU-95-03, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta GA, Jan. 1995.

[17] W. Meira Jr., T.J. LeBlanc and A. Poulos, *Waiting Time Analysis and Performance Visualization in Carnival*, ACM SIGMETRICS Symp. on Par. and Dist. Tools, 1-10, May 1996.

[18] J. Kohn and W. Williams, *ATExpert*, J. Par. and Dist. Comp. **18**, 205-222, 1993.