HydraMP: Exploiting shared memory parallelism in HYDRA with OpenMP

Dingle, Nicholas

2011

MIMS EPrint: **2011.62**

ISSN 1749-9097

HydraMP: Exploiting shared memory parallelism in HYDRA with OpenMP

Nicholas J. Dingle

School of Mathematics, University of Manchester, Manchester, M13 9PL
nicholas.dingle@manchester.ac.uk

Abstract. Multicore CPUs are now found in desktops, servers and supercomputers but many existing parallel performance analysis tools were designed for the single-core distributed-memory world. In this paper we investigate the practicality of taking an existing tool, namely the HYDRA response time analyser, and parallelising it with OpenMP to produce a multi-threaded implementation suitable for execution on multicore shared-memory machines. We discuss the amount of software engineering work required to do this, and show that only a small number of lines of code need to be added to achieve dramatic speed-ups over the serial version. We also compare the run-times of our OpenMP-parallelised version with existing MPI-parallelised code on the same hardware.

1 Introduction

It is a truth universally acknowledged that, for the foreseeable future at least, improvements to CPUs will focus on increasing the number of cores that they contain rather than increasing the cores' raw clock-speeds. This presents a new set of problems to software writers, and those working in the performance analysis domain are not immune. Previously, performance improvements to existing code were granted simply by the new generation of single-core CPUs running faster than the previous generation, but exploiting the extra computing power of multicore CPUs is a much more complicated task that requires existing serial code to be rewritten to take advantage of opportunities for parallelism.

Many existing parallel performance analysis tools, for example DNAmaca [1, 2] and HYDRA [3, 4], date from turn of the century when single-core processors were the norm. Multicore versions of these tools, where they exist, were typically written to run on distributed-memory nodes that each contained a single core, necessitating the exchange of data across the network by means of libraries such as Message Passing Interface (MPI) [5] or Parallel Virtual Machine (PVM) [6]. Running these MPI-based solvers on a single modern shared-memory multicore machine is possible but unwieldy: the built-in assumption that memory is distributed usually means that there is a data-partitioning stage that is unnecessary on a shared-memory machine. The user must also rely on the underlying communication library to implement shared-memory operations efficiently.

Recent advances in shared memory parallel programming libraries such as OpenMP [7] give an easy way to parallelise existing serial solvers for shared memory environments. They are typically easier to incorporate into existing code than MPI as they do not require the programmer to explicitly code in the work-distribution and interprocessor-communication steps. This raises the possibility that it might be feasible to achieve good speed-ups of existing single-threaded tools such as HYDRA on modern multicore CPUs for little programming effort.

In this paper we investigate the speed-ups achieved in parallelising the existing performance tool HYDRA using OpenMP. There are two opportunities for parallelism in this code: the repeated sparse matrix–vector multiplications and the calculation of Erlang distribution terms. We consider the effort required to parallelise both of these using OpenMP, and show the effect that each has on the overall run-time of the tool. We also compare the resulting OpenMP-parallelised version of HYDRA, which we call HydraMP, with the serial HYDRA and with the MPI-parallelised HYDRA running on a single multicore machine, and highlight the run-time speed-ups achieved.

The remainder of this paper is organised as follows. Section 2 describes OpenMP and the HYDRA tool. The parallelisation of sparse matrix–vector multiplication is addressed in Section 3, where we discuss how this is implemented in HYDRA and compare the performance of the OpenMP-parallelised implementation with that of the original MPI-parallelised HYDRA. Section 4 then discusses how HYDRA's calculation of the Erlang distribution terms can be parallelised and presents results showing the performance improvement of so doing compared with serial HYDRA. In Section 5 we look at the overall effect of combining the two forms of OpenMP parallelisation in a single version of HYDRA. Section 6 discusses related work and finally Section 7 concludes and discusses opportunities for future work.

2 Background

This section presents a brief overview of OpenMP (Open Multi-Processing) [7] and then discusses the uniformisation technique for calculating full response time densities and distributions in Continuous Time Markov Chains (CTMCs) before summarising its implementation in the HYpergraph-based Distributed Response-time Analyser (HYDRA) [3, 4].

2.1 OpenMP

OpenMP is a multithreading extension for C/C++ and Fortran that allows users to easily exploit shared-memory parallelism on a range of modern computing architectures. It follows a fork-join model, with slave threads being spawned from a single master thread on demand. These threads execute specified code sections in parallel before synchronising at the end and returning control to the master thread.

OpenMP includes a number of work-sharing constructs. For the purposes of this paper the main one is `omp for`, which divides loop iterations between participating threads. Unless explicitly specified by the programmer there will be no fixed ordering of the iterations, which means that correctness will only be maintained if this is applied to a loop in which there is no dependence between iterations; the work done in each loop iteration must not rely on results computed in a previous (lower-numbered) iteration. OpenMP also has the `omp sections` construct, which allows the programmer to specify different blocks of code to be executed by each participating thread. The work presented in this paper does not exploit `sections`, and so we do not discuss it further.

The process of creating threads, dividing work between them and destroying them when they are no longer required is automatically handled by the compiler and run-time environment. This is in contrast to libraries like MPI, where the programmer must encode not only the program logic but also the division of data and the pattern of inter-processor communication.

```
#pragma omp parallel for
for (int n=0; n<10; n++) {
  printf("%i ", n);
}
```

Fig. 1. Specifying a parallel loop in OpenMP.

In both C/C++ and Fortran, OpenMP programming is achieved through the use of compiler directives inserted into the source-code. For example, Fig. 1 shows a fragment of C code including the use of the OpenMP directive `#pragma omp parallel for` to parallelise a simple loop printing out the numbers 0 to 9. In the serial case this loop will print out the numbers in-order, but when multiple threads are used this will no longer be the case.

It will be seen that parallelising existing code with OpenMP can be very straightforward: if the program contains suitable constructs it is easy to convert them into parallel constructs and this can lead to much improved performance. Sometimes, however, further modifications may be needed to ensure correctness is maintained and that good multithreaded performance is achieved.

2.2 Uniformisation

Response time densities and quantiles in CTMCs can be computed through the use of uniformisation (also known as randomization) [8–11]. This transforms a CTMC into one in which all states have the same mean holding time, $1/q$, by allowing "invisible" transitions from a state to itself. This is equivalent to a discrete-time Markov chain, after normalisation of the rows, together with an associated Poisson process of rate q. The one-step transition probability matrix $\mathbf{P}$ which characterises the one-step behaviour of the uniformised DTMC is derived from the generator matrix $\mathbf{Q}$ of the CTMC as:

$$\mathbf{P} = \mathbf{Q}/q + \mathbf{I} \tag{1}$$

where the rate $q > \max_i |q_{ii}|$ ensures that the DTMC is aperiodic.

The calculation of the first passage time density between two states has two main components. The first considers the time to complete n hops ($n = 1, 2, 3, \ldots$). Recall that in the uniformised chain all transitions occur with rate q. This means that the convolution of n of these holding-time densities is the convolution of n exponentials all with rate q, which is an n-stage Erlang density with rate q.

Secondly, it is necessary to calculate the probability that the transition between a source and target state occurs in exactly n hops of the uniformised chain, for every value of n between 1 and a maximum value m. This is calculated by repeated sparse matrix–vector multiplications. The value of m is determined when the value of the nth Erlang density function drops below a threshold value. After this point, further terms are deemed to add nothing significant to the passage time density and are disregarded.

The density of the time to pass between a source state i and a target state j in a uniformised Markov chain can therefore be expressed as the sum of m n-stage Erlang densities, weighted with the probability that the chain moves from state i to state j in exactly n hops ($1 \leq n \leq m$). The response time between the non-empty set of source states $\boldsymbol{i}$ and the non-empty set of target states $\boldsymbol{j}$ therefore has probability density function:

$$\begin{aligned} f_{\boldsymbol{i}\boldsymbol{j}}(t) &= \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \\ &\simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \end{aligned} \tag{2}$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P} \qquad \text{for } n \geq 0 \tag{3}$$

2.3 HYDRA

HYDRA is a performance analysis tool which implements the uniformisation technique to compute response time densities and distributions in CTMC models specified in a number of high-level modelling formalisms. HYDRA has proved to be particularly popular with modellers who work with the stochastic process algebra PEPA (Performance Evaluation Process Algebra) thanks to the interface provided by the International PEPA Compiler (ipc) [12] and has accordingly been used to analyse a range of CTMC models, including:

- Software systems [13], and more specifically assembly code [14], content adaptation systems [15] and a software retrieval service derived from a UML model [16],
- Wireless protocols [17],
- Timing attacks on communications protocols [18],
- Service Level Agreements (SLAs) [19, 20],
- Grid computing systems [21],
- Role-playing games [22].

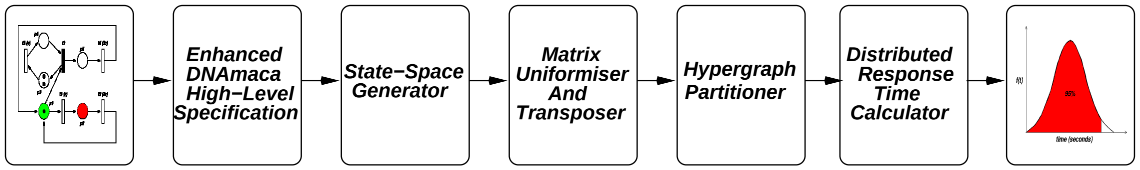
Enhanced DNAmaca High–Level Specification
State–Space Generator
Matrix Uniformiser And Transposer
Hypergraph Partitioner
Distributed Response Time Calculator
time (seconds)

Fig. 2. HYDRA tool architecture [4].

– Stochastic Well-formed Network models of healthcare organisations [23]

Fig. 2 shows the architecture of the HYDRA tool. The process of calculating a response time density begins with a high-level model specified in an enhanced form of the DNAmaca interface language. Next, a probabilistic, hash-based state generator uses the high-level model description to produce the generator matrix $\mathbf{Q}$ of the model's underlying Markov chain as well as a list of the initial and target states. $\mathbf{P}$ is constructed from $\mathbf{Q}$ according to Eq. 1.

The pipeline is completed by our response time calculator. Two versions of this exist, both implemented in C++: a serial implementation, and a distributed version which uses MPI to parallelise the sparse matrix–vector multiplications of Eq. 3. Both versions have essentially the same structure. First, the maximum number of hops m is calculated by computing the Erlang terms for highest value of t for which $f_{\boldsymbol{ij}}(t)$ is required. When these terms fall below a specified threshold value then the maximum number of hops is deemed to have been reached. The vector $\boldsymbol{\pi}^{(n)}$ is then calculated for $n = 1, 2, 3, \ldots, m$ by repeated sparse matrix–vector multiplications of the form of Eq. 3. In the parallel version of HYDRA it is necessary to map the non-zero elements of $\mathbf{P}$ onto processors such that the computational load is balanced and communication between processors is minimised if this computation is to be done efficiently. To achieve this, we use hypergraph partitioning to assign matrix rows and corresponding vector elements to processors [24]. The sum $\sum_{k \in \boldsymbol{j}} \pi_k^{(n)}$ is calculated from each vector $\boldsymbol{\pi}^{(n)}$.

The program then loops over all values of t for which $f_{\boldsymbol{ij}}(t)$ is required, calculating the corresponding Erlang density/distribution terms and multiplying them with the vector sums. In the parallel version of HYDRA this procedure only takes place on the master processor. The resulting points are written to a disk file and are displayed using the GNUplot graph plotting utility.

3 The Hard Bit: Sparse Matrix–Vector Multiplication

HYDRA uses DNAmaca's sparse matrix representation, summarised in Fig. 3, to store $\mathbf{P}$ in a memory-efficient fashion. Individual matrix elements are not stored as distinct `doubles`, but rather unique matrix element values are kept in the `Store` and the matrix maintains pointers into this `Store` for each element. The `AVLTree` is used to efficiently search the `Store` when new elements are added to the matrix. For matrices derived from high-level modelling formalisms such as Petri nets, this scheme can reduce the amount of memory required as the matrix entries will often be rates (or functions of rates) taken from the high-level model description, and these will often have far fewer distinct values than the number of states or transitions in the underlying CTMC.

Sparse matrix–vector multiplication in HYDRA is accomplished using the `SparseMatrix` class's `transMultiply()` method, which is summarised in Fig. 4. This loops over the columns of the matrix and multiplies each entry with the corresponding vector element, storing the result in the `result` vector. This method would seem to be a likely candidate for parallelisation with OpenMP as the iterations are independent of each-other. Doing this is as simple as adding two lines of code:

```
#include "omp.h"
```

goes at the beginning, and:

```
#pragma omp parallel for default(none) private(_col,t,sum) shared(x,result)
                              schedule(guided,1)
```

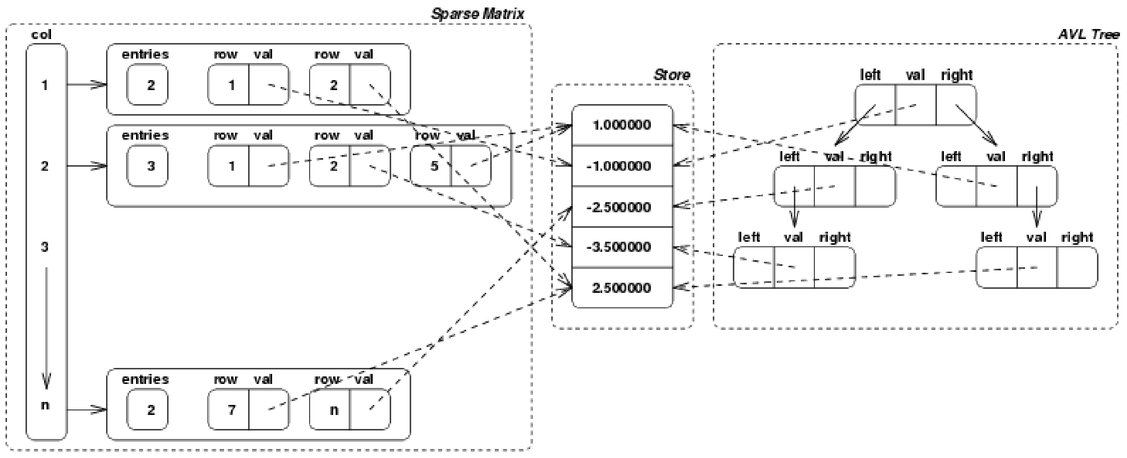

Fig. 3. DNAmaca's sparse matrix representation [1].

```
void SparseMatrix::transMultiply(Vector<double> &x, Vector <double> &result) const {
  double sum;
  long i,t;
  SparseMatrixCol *_col;

  for (i=0; i<n; i++) {
    _col = &col(i);
    sum = 0;
    for (t=0; t<_col->getEntries(); t++) {
      sum += (*(_col->getIndex(t)))*x(_col->getRow(t));
    }
    result[i] = sum;
  }
}
```

Fig. 4. The `transMultiply()` method from DNAmaca's `SparseMatrix` class for a matrix with n columns.

```
double data[nz];
long row[nz];
long offset[n+1];

double pi[n];
double result[n];

for (long i=0; i<n; i++) {
  double sum = 0;
  long begin = offset[i];
  long end = offset[i+1];

  for (long t=begin; t<end; t++) {
    sum += data[t] * pi[row[t]];
  }
  result[i] = sum;
}
```

Fig. 5. CSR representation and matrix–vector multiplication routine for a matrix with n rows and nz non-zero entries and a vector `pi` with n entries.

is placed before the loop indexed by `long i` in Fig. 4. The final addition is to tell the compiler to use OpenMP by means of the `-fopenmp` flag.

We contrast DNAmaca's `SparseMatrix` class with the well-known Compressed Sparse Row (CSR) [25] scheme. This stores the matrix in three arrays: one containing the matrix element values, one containing the column indices of each element and one containing the offsets into the first two arrays marking the start of each row. Matrix element values are stored explicitly rather than as pointers into a list of distinct values, however, which increases the amount of memory used compared with the `SparseMatrix` scheme. The advantages of CSR are that there are fewer indirect memory accesses and also more regular memory access patterns (as we are iterating over 1-dimensional arrays) which can lead to improved performance.

This CSR data structure and the code used to perform sparse matrix–vector multiplication with a vector `pi` is shown in Fig. 5. The multiplication can also be parallelised with OpenMP by placing a `#pragma omp parallel for` outside the loop indexed by `long i`.

Model	# States	Serial `transMultiply()`	Parallel `transMultiply()`				Parallel CSR			
			1	2	3	4	1	2	3	4
`courier`	11 700	0.35	0.36	0.20	0.14	0.11	0.38	0.21	0.14	0.12
`fms`	537 768	61.27	61.84	43.43	38.38	36.52	20.44	15.92	15.09	15.62
	1 639 440	202.64	202.83	142.15	127.55	119.98	70.76	54.90	53.85	53.33
	4 459 455	576.84	579.96	405.43	359.09	341.52	209.82	164.14	156.81	156.32

Table 1. Run-times in seconds for repeated sparse matrix–vector multiplication for `SparseMatrix` and CSR formats parallelised with 1 to 4 OpenMP threads.

Tab. 1 presents the time taken to perform 1000 sparse matrix–vector multiplications using the OpenMP parallelisation of `SparseMatrix::transMultiply()` and also the parallelised CSR for a range of matrix sizes and number of threads, and compares these results against the single-threaded performance of the original serial `SparseMatrix::transMultiply()`. Results were produced on an Intel Core2 3.0GHz quad-core CPU workstation with 8GB RAM using between 1 and 4 threads. Each entry in Tab. 1 is the average of 5 runs. The models used are the well-known Courier [26] and Flexible Manufacturing System (FMS) [27] Generalised Stochastic Petri nets (GSPNs).

It can be seen that both parallelisation schemes offer performance improvements over the serial `transMultiply()` in DNAmaca, but that parallel CSR is between 2 and 3 times faster than parallel `transMultiply()` in all but the smallest example (`courier`). This is to be expected, for the reasons we discussed above. The performance gained purely from moving from `SparseMatrix` to CSR format can be seen by comparing the run-times with only 1 OpenMP thread.

When it comes to deciding which of these schemes should be incorporated into HYDRA we are forced to trade off performance again convenience. Parallel `transMultiply()` is the simplest to implement as it requires little more than adding `#pragma omp parallel for` around the loop over `i` in Fig. 4. Switching HYDRA's storage format from the `SparseMatrix` class to CSR has the potential to result in even better performance but requires far more work: HYDRA currently exploits all the class methods for reading in the matrix and doing the multiplication, and all these would need to be rewritten to take advantage of CSR format.

3.1 Comparison with MPI-Parallelised Sparse Matrix–Vector Multiplication

As discussed above, the original HYDRA parallelises the sparse matrix–vector multiplications with MPI. It is interesting therefore to compare its run-time with that of the OpenMP-parallelised `transMultiply()` version on a shared-memory machine. Once again, we use an Intel Core2 3.0GHz quad-core CPU workstation with 8GB RAM.

Model	# States	# iterations	Serial	OpenMP	MPI (row-striped)	MPI (hypergraph)
`courier`	11 700	1 329	0.38	0.19	0.10	0.10
`fms`	537 768	712	26.4	16.4	15.1	13.9
	1 639 440	712	83.9	51.7	52.2	49.9
	4 459 455	712	234.5	142.7	154.9	152.5

Table 2. Run-times in seconds for serial, OpenMP and MPI sparse matrix–vector multiplication on a shared-memory machine. All parallel results used 4 threads/processes.

Tab. 2 presents the run-times for serial and parallel sparse matrix–vector multiplications for a range of model sizes and number of iterations. Note that each entry in the table is the average of 5 runs. We consider two data-partitioning schemes for the MPI-parallelised case: row-striping, where each processor is allocated a block of contiguous rows such that each processor has approximately the same number of non-zeros, and hypergraph, where computational load is again balanced but communication between processors is minimised. In a distributed-memory setting the latter scheme is very much more efficient than the former, and it is interesting to note that the same applies in a shared-memory setting. Presumably this is because hypergraph partitioning results in less copying of data between the processes' address spaces.

The results presented in Tab. 2 show that the performance of parallel sparse matrix–vector multiplication is essentially the same for OpenMP and MPI. When the matrix is small then using MPI is faster, but as the matrix size increases then OpenMP becomes the quicker of the two. We theorise that this can be attributed to the overhead of creating and destroying thread groups in OpenMP: when the matrix is small then this process dominates the solution time, but for larger matrices the start-up cost becomes insignificant compared to the time required to perform the actual computations.

In the cases where the MPI version of HYDRA is faster, we must acknowledge that it is significantly more complicated to use: it requires us to execute a separate partitioning program (possibly a third-party hypergraph partitioner) and to apply the results to the matrix before we can even begin to execute the response time analyser, while the OpenMP version is executed in exactly the same way as the serial version. It is entirely conceivable that the run-time improvements seen in the MPI results in Tab. 2 will be overshadowed by these extra data-partitioning stages. We therefore conclude that in a shared-memory environment the use of OpenMP to parallelise the sparse matrix–vector multiplications is preferable to the use of MPI.

4 The Easy Bit: Erlang Term Parallelisation

```
for (int count=0; count < (int)(t_top/t_step); count++) {

  double pdf_answer = 0.0;
  double cdf_answer = 0.0;

  for (int n=1; n <= m; n++) {
    pdf_answer += log_erlang(n, _q, (count+1)*t_step) * sum_pi_target[n];
    cdf_answer += log_erlang_cdf(n, _q, (count+1)*t_step) * sum_pi_target[n];
  }

  pdf_result_output << (count+1)*t_step << " " << pdf_answer << endl;
  cdf_result_output << (count+1)*t_step << " " << cdf_answer << endl;
}
```

Fig. 6. HYDRA loop for calculating $f_{ij}(t)$ at all required values of t.

HYDRA's loop over the values of t at which $f_{ij}(t)$ is required is summarised in Fig. 6. The inner loop (from $n = 1$ to m) corresponds to the outer summation in Eq. 2, while the `sum_pi_target` array holds the precomputed $\sum_{k \in j} \pi_k^{(n)}$ terms (the inner summation in Eq. 2) that are calculated by repeated sparse matrix–vector multiplication as discussed in the previous section. Note that the outer loop iterations are independent of each-other, which makes it an ideal candidate for OpenMP parallelisation. Doing this is again easily achieved by adding:

```
#pragma omp parallel for shared(max_hops,sum_pi_target) schedule(guided)
```

before the loop indexed by `int count` in Fig. 6.

Model	# States	# t-points	Serial Mx–Vec	Serial Erlang	Serial Time (s)	OpenMP Time (s)	Speed-up
`courier`	11 700	100	1.7%	97.1%	22.1	6.1	3.6
		1 000	0.2%	99.7%	216.5	55.3	3.9
		10 000	0.02%	99.9%	2 149.9	541.2	4.0
`fms`	537 768	100	79.9%	18.7%	33.1	28.3	1.1
		1 000	29.7%	69.7%	88.8	42.1	2.1
		10 000	4.0%	95.9%	647.0	182.0	3.6
	1 639 440	100	91.9%	6.8%	91.6	86.4	1.1
		1 000	57.2%	41.9%	147.2	100.5	1.5
		10 000	12.0%	87.9%	704.5	239.8	2.9
	4 459 455	100	95.9%	2.5%	245.8	238.9	1.0
		1 000	78.3%	20.5%	301.7	252.8	1.2
		10 000	27.6%	71.9%	858.2	392.3	2.2

Table 3. Run-times for serial and 4-thread OpenMP versions of the loop in Fig. 6.

The result of applying these seemingly minor changes can be seen in Tab. 3, which compares the run-time for the original serial version of HYDRA with that of the modified version which uses OpenMP with 4 threads. Results were produced on an Intel Core2 3.0GHz quad-core CPU workstation with 8GB RAM for a range of CTMC sizes and number of t-points. The corresponding speed-ups are also shown. Each entry in Tab. 3 is the average of 5 runs. Note that all matrix–vector multiplications here are conducted in serial; the effect of incorporating parallel matrix–vector multiplication will be considered in the next section.

As we would expect the largest speed-up that is observed is 4 as this problem is trivially parallelisable. The size of the speed-up across the different problem sizes and number of t-points obviously depends on the amount of work that can be parallelised; in those cases where the run-time is dominated by the time required to do the sparse matrix–vector multiplications, the improvement from speeding up the t-point loop is correspondingly limited. This is Amdahl's law in action [28, 29].

The number of t-points is independent of the size of the matrix and is instead specified by the user in the initial performance query. This means that in general it is hard to reason about the relative amount of work that the matrix–vector multiplications and the Erlang term calculation will require – for any model it depends entirely on the performance query being asked. The results in Tab. 3 suggest that parallelising the loop shown in Fig. 6 is worthwhile, however. For very little software engineering effort we can gain some speed-up in execution time, and the overhead from OpenMP seems to be sufficiently small that even in cases where there is limited opportunity for parallelism (e.g. in the 100 t-point case for the 4 459 455-state CTMC) we are no worse off.

One slight drawback of the OpenMP version is that each thread outputs the pdf/cdf value at a particular value of t as it is computed, but because the iterations are divided across the participating threads the overall output will be out of order. This is a minor problem, however, as it does not prevent GNUplot from displaying the corresponding graph correctly, and if text output in increasing order of t is required the data file can easily be sorted once execution is complete.

5 Putting It All Together

Model	# States	# t-points	Serial Time (s)	OpenMP Time (s)	Speed-up
`courier`	11 700	100	22.1	5.9	3.7
		1 000	216.5	54.3	4.0
		10 000	2 149.9	547.0	3.9
`fms`	537 768	100	33.1	18.5	1.8
		1 000	88.8	32.4	2.7
		10 000	647.0	171.3	3.8
	1 639 440	100	91.6	54.6	1.7
		1 000	147.2	68.6	2.1
		10 000	704.5	207.6	3.4
	4 459 455	100	245.8	148.2	1.7
		1 000	301.7	162.1	1.9
		10 000	858.2	301.6	2.8

Table 4. Run-times for serial HYDRA and 4-thread OpenMP version with both sparse matrix–vector multiplication and t-point loop parallelisation.

We now look at the effect of bringing the two parallelisation opportunities described in Sections 3 and 4 together in a single implementation. Table 4 compares the run-time of the original serial HYDRA with a parallel version that uses OpenMP to parallelise both the sparse matrix–vector multiplications and the t-point loop. Results were produced on an Intel Core2 3.0GHz quad-core CPU workstation with 8GB RAM for a range of CTMC sizes and number of t-points. The corresponding speed-ups are also shown. Each entry in Tab. 4 is the average of 5 runs.

What is striking from Tab. 4 is that for very little software engineering effort we have produced a version of HYDRA that runs between 1.7 and 4.0 times faster on a modern multicore desktop. The exact speed-up achieved depends on the problem size and number of t-points required, but in all cases we do observe a marked improvement for the incorporation of OpenMP into HYDRA.

6 Related Work

Sparse matrix–vector multiplication is the kernel of a wide range of scientific techniques, and as such its efficient implementation has been widely studied. Its performance is usually constrained by available memory bandwidth rather than instruction processing speed (see [30] for a fuller discussion), and this is exacerbated by the poor temporal locality of accesses to the vector which reduces the effectiveness of caches. There has therefore been a great deal of work on improved matrix storage formats to overcome these limitations, including numerous variants of Block CSR (BCSR) and matrix reordering [31–33].

There has similarly been a great deal of work on the performance of sparse matrix–vector multiplication on shared-memory architectures. The work in [34], which analyses performance on a variety processors including Intel, AMD and Cell, is particularly relevant as it includes a comparison of a Pthreads implementation with an MPI implementation. It is observed that the Pthreads version is more than twice as fast as the MPI version.

There has also been a number of papers investigating good matrix storage formats for OpenMP parallelised sparse matrix–vector multiplications. In [35] the authors investigate the relative performance of standard CSR versus BCSR under OpenMP's standard work partitioning schemes and their own load-balancing approach, and conclude that block partitioning that balances the number of non-zeros across participating processors gives the best performance. Similarly, [36] looks at the performance of sparse matrix–vector multiplication using OpenMP with CSR, BCSR and diagonal matrix formats on a range of architectures. The authors concluded that the best format was architecture dependent.

Based on this existing literature, we conclude that if we want to achieve the best possible performance for parallel sparse matrix–vector multiplication we should investigate storage formats other than those considered in this paper. Implementing block storage schemes in HYDRA would require a major rewrite of the existing code, however, and must therefore be a topic for future work: the focus of the current paper is on the benefits that can be gained with as little modification to the existing program as possible.

7 Conclusion

We have taken the existing performance tool HYDRA and investigated how its core calculations can be parallelised for modern multicore processors by using OpenMP. We have shown that there are two opportunities for parallelism, namely in the repeated sparse matrix–vector multiplications and in calculating the values of $f_{ij}(t)$ for each required value of t, and that in both cases the use of OpenMP can give performance improvements over serial HYDRA at the cost of adding a 3 or 4 extra lines of code. To give some idea of the relative scale of effort versus developing the original software, the serial version of HYDRA (including all headers and class files) comprises over 12 000 lines of code. If further code refactoring work is acceptable then replacing the existing `SparseMatrix` class from DNAmaca with a matrix stored in a more efficient format has the potential to yield even further reductions in run-time.

There are a number of opportunities for further work. This paper has focused entirely on the final stage in the HYDRA pipeline of Fig. 2, but there has been other work on parallelising the earlier stages (particularly the state generation and steady-state solution phases) for distributed-memory machines by using MPI [2]. It would be interesting to see if these stages are also suitable for parallelisation with OpenMP.

Finally, it should be noted that MPI and OpenMP are not mutually exclusive. Multicore machines form the building-blocks of almost all modern clusters and supercomputers, and so there is scope for a hybrid implementation of HYDRA where OpenMP is used to parallelise the calculations within the participating nodes and MPI is used to parallelise the problem across multiple nodes.

Acknowledgements

The author would like to acknowledge the assistance given by IT Services for Research (formerly Research Computing Services) at the University of Manchester, particularly regarding the use of OpenMP.

References

1. Knottenbelt, W.: Generalised Markovian analysis of timed transition systems. Master's thesis, Department of Computer Science, University of Cape Town (1996)
2. Knottenbelt, W.: Parallel Performance Analysis of Large Markov Models. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine (1999)
3. Dingle, N., Harrison, P., Knottenbelt, W.: Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. Journal of Parallel and Distributed Computing **64** (2004) 908–920
4. Dingle, N., Harrison, P., Knottenbelt, W.: HYDRA: HYpergraph-based Distributed Response-time Analyser. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), Las Vegas NV, USA (2003) 215–219
5. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. Second edn. MIT Press, Cambridge, Massachusetts (1999)
6. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
7. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, Cambridge, Massachusetts (2007)
8. Bolch, G., Greiner, S., Meer, H., Trivedi, K.: Queueing Networks and Markov Chains. Wiley (1998)
9. Melamed, B., Yadin, M.: Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes. Operations Research **32** (1984) 926–944
10. Miner, A.: Computing response time distributions using stochastic Petri nets and matrix diagrams. In: Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM'03), Urbana-Champaign, IL (2003) 10–19
11. Muppala, J., Trivedi, K.: Numerical transient analysis of finite Markovian queueing systems. In Bhat, U., Basawa, I., eds.: Queueing and Related Models, Oxford University Press (1992) 262–284
12. Bradley, J., Knottenbelt, W.: The ipc/HYDRA tool chain for the analysis of PEPA models. In: Proceedings of the 1st IEEE International Conference on the Quantitative Evaluation of Systems. (2004) 334–335
13. Gilmore, S., Hillston, J., Kloul, L., Ribaudo, M.: Software performance modelling using PEPA nets. In: Proceedings of the 4th international Workshop on Software and Performance (WOSP'04). (2004) 13–23
14. Djoudi, L., Kloul, L.: Assembly code analysis using stochastic process algebra. In Thomas, N., Juiz, C., eds.: Computer Performance Engineering. Volume 5261 of Lecture Notes in Computer Science. (2008) 95–109
15. Ding, J.: Structural and Fluid Analysis for Large Scale PEPA Models – With Applications to Content Adaptation Systems. PhD thesis, University of Edinburgh (2010)
16. Kloul, L.: Performance analysis of a software retrieval service. Electronic Notes in Theoretical Computer Science **232** (2009) 145 – 163 Proceedings of the Third International Workshop on the Practical Application of Stochastic Modelling (PASM'08).
17. Argent-Katwala, A., Bradley, J., Geisweiller, N., Gilmore, S., Thomas, N.: Modelling tools and techniques for the performance analysis of wireless protocols. In Ming, G., Pan, Y., Fan, P., eds.: Advances in Wireless Networks: Performance Modelling, Analysis and Enhancement. Nova Science Publishers, Inc (2007) 3–39
18. Buchholtz, M., Gilmore, S., Hillston, J., Nielson, F.: Securing statically-verified communications protocols against timing attacks. Electronic Notes in Theoretical Computer Science **128** (2005) 123–143 Proceedings of the First International Workshop on Practical Applications of Stochastic Modelling (PASM'04).
19. Clark, A., Gilmore, S.: Evaluating quality of service for service level agreements. In Brim, L., Haverkort, B., Leucker, M., van de Pol, J., eds.: Formal Methods: Applications and Technology. Volume 4346 of Lecture Notes in Computer Science. (2007) 181–194

20. Clark, A., Gilmore, S., Tribastone, M.: Service-level agreements for service-oriented computing. In Corradini, A., Montanari, U., eds.: Recent Trends in Algebraic Development Techniques. Volume 5486 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 21–36
21. Thomas, N., Bradley, J., Knottenbelt, W.: Stochastic analysis of scheduling strategies in a Grid-based resource model. Software, IEE Proceedings - **151** (2004) 232–239
22. Gilmore, S., Kloul, L., Piazza, D.: Modelling role-playing games using PEPA nets. In Aykanat, C., Dayar, T., Körpeoglu, I., eds.: Computer and Information Sciences – ISCIS 2004. Volume 3280 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 523–532
23. Balbo, G., Beccuti, M., Pierro, M.D., Franceschinis, G.: Computing first passage time distributions in Stochastic Well-formed Nets. In: Proceeding of the 2nd joint WOSP/SIPEW International Conference on Performance Engineering (ICPE'11). (2011) 7–18
24. Catalyürek, U., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 673–693
25. Duff, I., Erisman, A., Reid, J.: Direct Methods for Sparse Matrices. Clarendon Press, Oxford (1986)
26. Woodside, C., Li, Y.: Performance Petri net analysis of communication protocol software by delay-equivalent aggregation. In: Proceedings of the 4th International Workshop on Petri nets and Performance Models (PNPM'91), Melbourne, Australia (1991) 64–73
27. Ciardo, G., Trivedi, K.: A decomposition approach for stochastic reward net models. Performance Evaluation **18** (1993) 37–59
28. Amdahl, G.: Validity of the single processor approach in achieving large scale computing capabilities. In: AFIPS Conference Proceedings. (1967) 483–485
29. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Second edn. Pearson Eduction Ltd. (2003)
30. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J. Supercomput. **50** (2009) 36–77
31. Im, E., Yelick, K.: Optimizing sparse matrix vector multiplication on SMPs. In: Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing (PPSC'99). (1999)
32. Pinar, A., Heath, M.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE conference on Supercomputing. (1999)
33. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research and Development **41** (1997) 711–725
34. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. (2007) 38:1–38:12
35. Liu, S., Zhang, Y., Sun, X., Qiu, R.: Performance evaluation of multithreaded sparse matrix-vector multiplication using OpenMP. In: Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC'09). (2009) 659–665
36. Kotakemori, H., Hasegawa, H.: Performance evaluation of a parallel iterative method library using OpenMP. In: Proceedings of the 8th International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05). (2005)