A Performance Estimator for Parallel Programs

Jeff Reeve

The Department of Electronics and Computer Science
The University of Southampton
Southampton SO16 1BJ
UK
jsr@ecs.soton.ac.uk

Abstract. In this paper we describe a Parallel Performance Estimator suitable for the comparative evaluation of parallel algorithms. The Estimator is designed for SPMD programs written in either C or FORTRAN. Simulation is used to produce estimates of execution times for varying numbers of processors and to analyse the communication overheads. Results from the estimator are compared with actual results (obtained on a 16 processor IBM SP2 machine) for an Alternating Direction Implicit (ADI) solver of linear equations and for a Parallel Sort by Regular Sampling (PSRS) sorting program. In both cases the plots of *Execution Time* versus *Number of Processors* are accurate to 20% and show all of the features of the equivalent plots of the measured data.

1 Introduction

Software providers with large production codes have always been concerned with measuring and enhancing the performance of their codes, since a faster, more memory efficient code is commercially advantageous. Even so there has never been a performance prediction tool available, even for sequential codes, that can reliably predict absolute performance. Consequently code optimisation is done by hand for particular codes using particular compilers and running on particular machines.

With parallel codes the problem might seem worse but if one settles for knowing the relative merits of one parallelisation strategy compared with another before embarking on a full scale parallel version of a code then considerable effort can be saved. In view of this our philosophy is to ignore the details of the sequential code and effect a simulation scheme that reproduces the communication pattern of the code on simplified architectures so that the speed of different parallel code versions can be assessed. This approach can also detect deadlock and provide information on the optimal scheduling for small embedded systems.

Our simulator accepts C or FORTRAN 77 source which contains calls to a message passing library. This means that and communication pattern can be accommodated.

Other approaches to the problem of estimating the performance of parallel programs have been for specific problems like data partitioning[3], or have taken a statistical approach[10,4] or have used static information only[5,6,8]. Other simulation based approaches that have some commonality with ours are by Kubota et al[14], Aversa et al[2] and Kempf et al[12].

2 Description

The data flow diagram for the Parallel Performance Estimator is shown in Figure 1. Each of the contributing components is explained below.

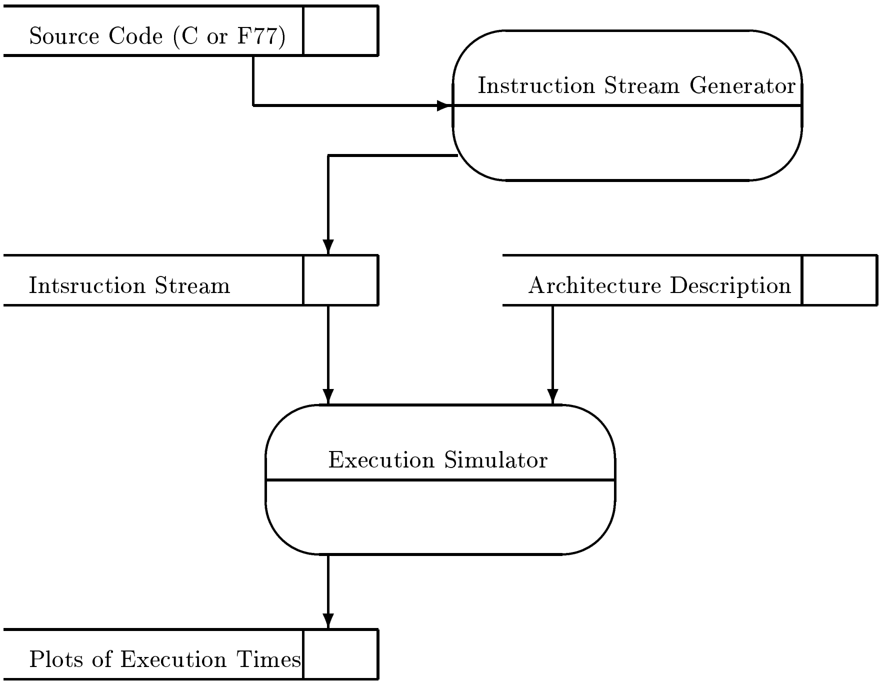

Fig. 1. The Data Flow Structure for the Performance Estimator

The Input Code

The source file input into the code generator takes the form of a F77 or C subroutine with input parameters consisting of the total number of processors (NPROCS) and the processor number (PID).

As this tools is designed to assist with algorithm development and not to analyse existing codes we have restricted the simulated instruction stream to a minimal set, that describe communication and computation events.

These allowed instructions that are accessed through the following subroutine calls are:

1. SEND(PROCESSOR_ID, NO_OF_WORDS)
   This generates a *non-blocking* send instruction of NO_OF_WORDS words to processor PROCESSOR_ID.
2. RECEIVE(PROCESSOR_ID, NO_OF_WORDS)
   This generates a *non-blocking* receive instruction of NO_OF_WORDS words from processor PROCESSOR_ID.
3. BSEND(PROCESSOR_ID, NO_OF_WORDS)
   This generates a send *blocking* instruction of NO_OF_WORDS words to processor PROCESSOR_ID.
4. BRECEIVE(PROCESSOR_ID, NO_OF_WORDS) This generates a *blocking* receive instruction of NO_OF_WORDS words from processor PROCESSOR_ID.
5. WORK(EQUIVALENT_NO_OF_FP_MULTIPLIES)
   This generates a work instruction which takes
   EQUIVALENT_NO_OF _FP_MULTIPLIES times
   TIME_OF_A_FP_MULTIPLY time.
6. WAIT() This generates a wait instruction which causes the simulator to wait for the completion of all outstanding communications events relating to the issueing processor before scheduling any more instructions for that processor.

This minimal set of six instructions is capable of representing almost all of the current flavours of communications schemes. Translation to MPI or PVM is relatively easy. Higher order functionality like *reduction* and *broadcast* routines are implemented using the minimal set using a spanning out tree through the network as the underlying topology for the reduction and distribution of collective data.

For existing codes a tool such as PERFORM[7] could be used to determine the complexity of the algorithm.

The Instruction Stream Generator

The code generator then incorporates the source and executes a driver that calls it successively for values of PID from 0 to NPROCS-1 and records an instruction stream for each of the processors.

The Architecture Description File

The architecture file contains the basic information needed to keep real time in the execution simulation. An example, for our test platform, a 16 processor IBM SP2, gleaned from benchmark results[11], is given below. There are two choices for the network architecture. The type 'bus' is used for single bus multiprocessor architectures such as ethernet systems, and the type 'nobus' is used for arctitectures in which the nodes are effectively equidistant, which is the case for most purpose built MPP systems.

```
SP2
receive latency = 0.24e-4 seconds
send latency = 0.24e-4 seconds
time to send 1 word (8bytes) = 2.39e-7 seconds
time for one multiply = 0.178e-7 seconds
network type = nobus
```

The Execution Simulator

The simulator interprets the instruction stream file, scheduling work and messages in an event queue, when the queue is empty and the instruction stream is exhausted then the program has successfully terminated. If the event queue is empty and the instruction stream isn't empty then the program has deadlocked. Should it occur, this condition is reported and the simulation terminates gracefully.

The initial action of the simulator is to issue a load event for each processor. From this point on the simulator is completely event driven. Each new event is scheduled with a time stamp. The earliest event on the event queue is serviced in turn. The cost of each instruction is determined from the architecture file.

For every architecture it is assumed that the system software is single threaded and offers only one entry point for the send and receive instructions. This means that every send and receive issued generates an event that is stamped with the current time extended by the appropriate send or receive latency time, and it is only when this event is serviced that the corresponding transmission event is put onto a separate queue in readiness for scheduling when both the processors participating in the event are ready and the bus (if appropriate) is ready.

The simulator keeps two queues. One is the event queue ordered in user program execution time, from which events are dequeued and acted on. The other is a queue of potential communications events. When the conditions for a communication are satisfied, ie both participants are ready and the data path is free, then the communication is inserted into the event queue. An outline of the simulator's event descheduler is given below.

```
SWITCH(EVENT_NAME)
   CASE LOAD
      SCHEDULE_NEXT_INSTRUCTION(PROCESS)
   CASE SEND_LATENCY | BSEND_LATENCY |
         RECEIVE_LATENCY | BRECEIVE_LATENCY
      SCHEDULE_NEXT_COMMUNICATION
      SCHEDULE_NEXT_INSTRUCTION(PROCESS)
      ALTER_STATE(PROCESS)
   CASE WORK
      SCHEDULE_NEXT_INSTRUCTION(PROCESS)
   CASE SEND_RECEIVE | BSEND_RECEIVE |
         SEND_BRECEIVE | BSEND_BRECEIVE
      FREE_BUS
```

```
SCHEDULE_NEXT_INSTRUCTION(SENDING PROCESS)
SCHEDULE_NEXT_INSTRUCTION(RECEIVING PROCESS)
SCHEDULE_NEXT_COMMUNICATION
UNBLOCK - SENDING/RECEIVING PROCESS INSTRUCTION STREAMS
```

3 Evaluation

We have chosen two exemplar codes to evaluate the Parallel Performance Estimator, the first is an Alternating Direction Implicit (ADI) two dimensional solver of initial value problems[17] and the second is a Parallel Sort by Regular Sampling (PSRS)[16]. In both cases we simulate the performance on our 16 node SP2 and compare the resulting times with actual run times.

The ADI Solver

The ADI solver is rich in communications structure, and behaves quite differently depending on the how the processor array is mapped onto the physical grid.

The ADI solution sequence starts with all of the processors in the first "Y-column", initiating forward elimination in the X direction. When a processor boundary is reached the data is passed to the processor on the right. Once this message is gone then another line of forward elimination in the X direction can be started. The process is repeated for the Y direction. The next step is an edge swap so that each process gets the updated halo data. After this is completed a *reduction* routine is used to determine the termination condition. For this operation the residual from each processor is sent to the bottom left hand corner of the processor array, the worst residual is determined at each node on the gather tree and the result sent to the parent. The root node send the worst residual back down the tree via its own daughters.

```
ADI
    FOR(T=0;T<T;T++)
        UNTIL CONVERGENCE
            FORWARD ELLIMINATE IN THE X DIRECTION
            BACK SUBSTITUTE IN THE X DIRECTION
            SWAP EDGE DATA
            FORWARD ELLIMINATE IN THE Y DIRECTION
            BACK SUBSTITUTE IN THE Y DIRECTION
            SWAP EDGE DATA
            DETERMINE THE CONVERGENCE BY REDUCING THE ERROR
```

The results of this verification are shown in Figures 2 and 3. The two test cases are for a 200×200 system of equations and a 100×100 system. As can be seen from graphs, the predicted timings are within 20% of the actual results, but more encouraging is the consistent accuracy of the prediction over a large number of processors. Figure 3 also shows how the simulator can break up the timings into its constituent parts. The sawtooth structure of the predicted values is due to the fact that the processor array changes configuration. For example for a prime number of processors Np the processors are in a $1 \times Np$ array. In general the processor array is $L_1 \times L_2 = Np$ where $L_1 \leq L_2$ and there is no other pair (L'_1, L'_2) that satisfies these conditions, for which $L'_1 > L_1$.

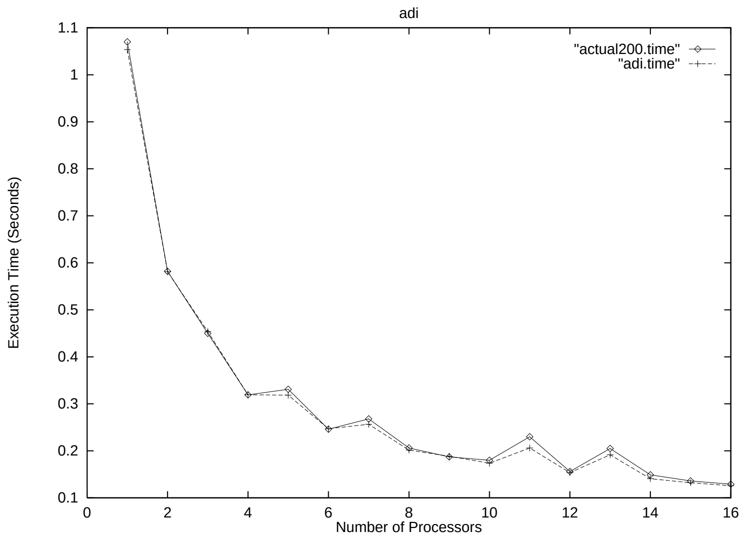

Fig. 2. Actual and Predicted Results Compared for a 200×200 System of Equations Solved by the ADI Method

The effect of the change in processor array is seen by the fact that the most inefficient solution is for a prime number of processors.

Parallel Sorting by Regular Sampling

This is a very effective parallel sorting algorithm developed by Li[15] with its principle attraction being that the number of sorted elements that end up on

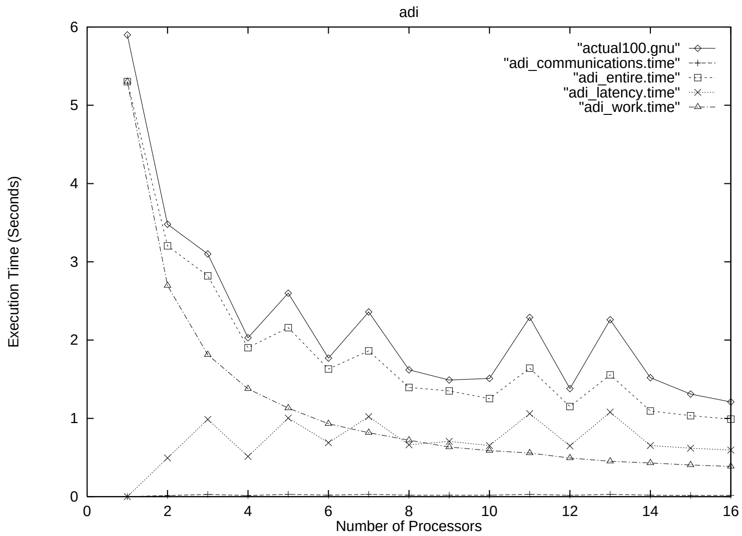

Fig. 3. Analysis of Actual and Predicted Results Compared for a 100×100 System of Equations Solved by the ADI Method

each processor is bounded and in practise for random initial data the variation in the number of sorted elements per processor is only $2-4\%$.

Assume that there are n elements to sort and that the number of processors p divides n without remainder. Initially each processor has $\frac{n}{p}$ of the elements. Each processor uses *quicksort* to order its elements and selects a sample of the elements at indices $1, (n/p^2)+1, (2n/p^2)+1, \cdots, (p-1)(n/p^2)+1$. These samples are sent to one processor which orders them and selects $p-1$ pivot values at indices $p+p/2, 2p+p/2, \cdots, (p-1)p+p/2$. These pivot values are sent to each processor which then divides its list into p parts identified by the pivot values. Now the ith processor keeps its ith partition and sends its jth partition to processor j. Each processor then merges its p partitions and the complete set is ordered accross the processors.

The pseudocode for the PSRS algorithm is given below.

```
PSRS
    GENERATE AND DISTRIBUTE THE INITIAL VALUES
    EACH PROCESSOR QUICKSORTS ITS ELEMENTS
    EACH PROCESSOR SELECTS ITS SAMPLES
```

THE SAMPLES ARE SENT TO THE MASTER PROCESSOR
THE MASTER PROCESSOR SELECTS AND BROADCASTS THE PIVOTS
EACH PROCESSOR PARTITIONS ITS LOCAL LIST
PARTITIONS ARE DISTRIBUTED AMONG PROCESSORS
EACH PROCESSOR MERGES ITS LIST

The simulation of this sorting method was compared with actual results for data sets of $1,000,000$ and $10,000$ real numbers respectively. The results are shown in Figures 4 and 5 respectively. Again the results are accurate to within 20%.

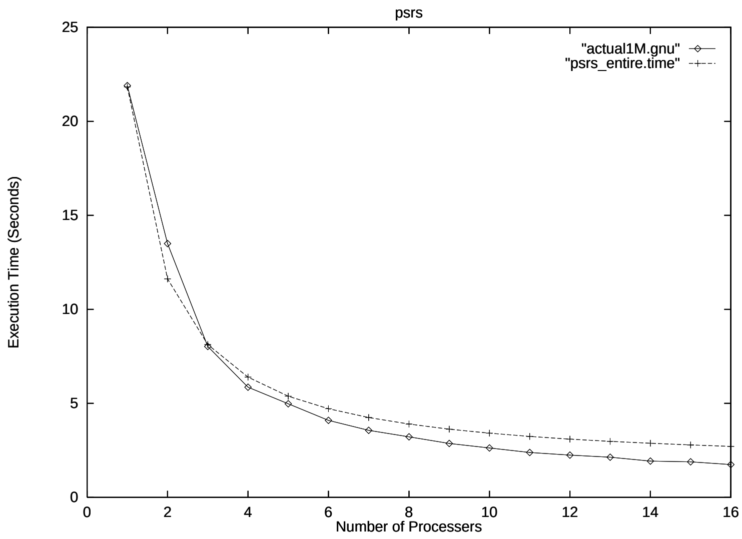

Fig. 4. Actual and Predicted Results Compared for 10^6 Doubles Sorted by PSRS

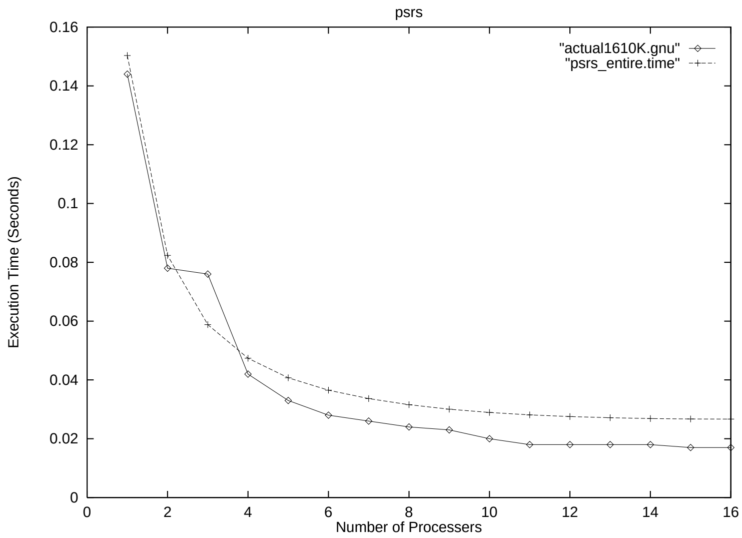

Fig. 5. Actual and Predicted Results Compared for 10^4 Doubles Sorted by PSRS

4 Conclusion

Both sets of test results that we have used show similar trends. While all tests show that the simulation replicates the broad features of the measured data it is evident that the simulation is the more accurate when communications is much less important than the work being done. There are we believe two causes for this. The first is that the single processor memory hierachy[18, 9, 1] (cacheing) is becoming important and the second is that the detailed working of the interconnect network is beginning to show. The cache effect can be modelled by introducing structure to the memory access times, but this introduces parameters that are difficult to measure. The network architecture could be simulated[13] but we believe that this effect is secondary compared with the contribution of the memory hierachy. Future work will concentrate on incorporating memorachy hierachy effects without introducing extra parameters.

Finally it is evident from the results of the tests that our Parallel Performance Estimator works effectively for both numerically intensive applications and for applications in which the movement of data dominates.

References

1. B. Alpern and L. Carter. Towards a model for portable parallel performance: Exposing the memory hierarchy. In A. Hey and J. Ferrante., editors, *Portability and Performance for Parallel Processing.*, pages 21–41, 1994.
2. R. Aversa, A. Mazzeo, N. Mazzocca, and U. Villano. Developing applications for heterogeneous computing environments using simulation: A case study. *Parallel Computing*, 24(5-6):741–762, 1998.
3. V. Balasundaram, V. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. *SIGPLAN Notices*, 26(7):213–223, 1991.
4. E. Brewer. High-level optimization via automated statistical modelling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, 1995.
5. M. Clement and M. Quinn. Architectural scaling and analytical performance prediction. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems.*, pages 16–21, 1994.
6. C.N. Dellarocas, E.A. Brewer, A. Colebrook, and W.E. Weihl. Proteus: A high -performance parallel algorithm evaluation system. Technical report, Dept Computer Science, MIT, 1991.
7. A.N. Dunlop, E. Hernandez, O.Naim, T. Hey, and D. Nicole. A toolkit for optimising parallel performance. *Lecture Notes in Computer Science.*, 919:548–553, 1995.
8. T. Fahringer. Compile-time estimation of communication costs for data parallel programs. *Journal of Parallel and Distributed Computing*, 39(1):46–65, 1996.
9. V.S. Getov. Performance characterisation of the cache memory effect. *Supercomputer*, 11:31–49, 1995.
10. F. Hartleb and V. Mertsiotakis. Bounds for the mean runtime of parallel programs. In *Sixth International Conference on Modelling Techniques and Tools for Parallel Computer Performance Evaluation.*, 1992.
11. E. Hernandez and T. Hey. Variations on low-level communication benchmarks. *Supercomputer*, 1997.
12. G. Kempf, A.J. van der Steen, C. Caremoli, and W.Y. Thang. Simulations of scientific programs on parallel architectures. *Lecture Notes In Computer Science*, 1067:536–546, 1996.
13. O. Kolp. Performance estimation for a parallel system with a heirachical switch network. *Parallel Computing.*, 20:1613–1626, 1994.
14. K. Kubota, K. Itakura, M. Sato, and T. Boku. Practical simulation of large-scale parallel programs and its performance analysis of the nas parallel benchmarks. *Lecture Notes in Computer Science*, 1470:244–254, 1998.
15. X. Li, P. Lu, J. Schaeffer, P.S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. Technical Report TR91-06, The Department of Computer Science, University of Alberta, March 1992.
16. M.Quinn. *Parallel Computing: Theory and Practice*, chapter 10. McGraw-Hill, Inc, 1994.
17. W.H. Press, B.P.Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recepies*, chapter 17. Cambridge University Press, 1986.
18. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the ACM SIGMETRICS conference on Measurement and Modelling of Computer Systems*, 1994.