PHYS52015 Coursework - Part1

Ciaran Reed
Core Ib: Introduction to Scientific and High-Performance Computing

This paper presents the parallelisation of a two-dimensional reaction-diffusion system using OpenMP. Strong scaling performance was evaluated across a range of core counts, with an estimated serial fraction of $f = 0.48$, indicating substantial parallel efficiency and identifying the evaluation of *dxdt* as the primary computational bottleneck.

I. Introduction

A serial codebase for simulating the reaction-diffusion system was provided, containing *init*, *dxdt*, *step*, *norm*, and *main*. OpenMP directives were added to parallelise the first four functions, and performance was assessed via strong scaling: $S(p) = t(1,1)/t(1,p)$ and efficiency $E(p) = S(p)/p$. If a fraction $0 < f < 1$ of the program is inherently serial, Amdahl's Law gives $t(n,p) = t(n,1)f + t(n,1)(1-f)/p$, yielding $S(p) = 1/(f + (1-f)/p)$; perfect scaling ($f \to 0$) is linear.

II. Methods

Each function was first implemented in a separate source file to evaluate runtime; results were later combined into a single program. Naively, one might assume that the common nested-for-loop structure in each function would allow the same parallelisation strategy to be applied uniformly; however, this is unlikely. The functions *init* and *step* are similar on the surface, whereas *norm* takes the form of a reduction operation. The simplest approach to parallelising these functions is to wrap the outer loop in a *OMP for* statement, which can be extended by adding *collapse(2)*. Alternatively, given the one-dimensional nature of the *u* and *v* arrays, the nested loops can be manually collapsed into a single loop. These three approaches were tested on *init*, *step*, and *norm*.

The function *dxdt* offers more opportunities for performance optimisation. Boundary points were handled separately from interior points to enable parallelization of the boundaries. Interior points can again be parallelised trivially, either with *collapse(2)* or by manually collapsing the iteration.

For each approach, the wall-clock runtime of the entire program was measured for core counts ranging from 1 to 64. By fitting the strong-scaling speedup to the recorded speedup, an estimate for the serial fraction f was obtained and used to compare results.

III. Results and Discussion

For *init*, the estimated serial fraction f for each implementation was: trivial = 1, collapse = 0.79, and manual = 0.87. It is somewhat surprising that the trivial implementation led to slowdowns; this can be attributed to insufficient parallelism in the outer loop, which leaves some threads idle. Each iteration requires two evaluations of *tanh*, making the function compute-bound. Introducing *collapse(2)* increases the available parallelism and improves load balancing, resulting in observable speedups. The compiler-managed collapse consistently outperformed the manual flattening approach, likely due to superior scheduling and optimization decisions by the compiler.

For *step*, the estimated serial fraction was $f = 1$ for all three parallel implementations, indicating slowdowns. Each iteration performs four loads (*du*, *dv*, *u*, *v*) and two stores (*u*, *v*) with minimal arithmetic, making this function memory bandwidth-bound. A single thread likely saturates the memory bandwidth, and additional threads compete for the same shared memory, resulting in no visible performance gains. Any overhead from creating the parallel region further degrades performance.

For *norm*, the estimated serial fraction f for each implementation was: trivial = 0.82, collapse = 0.91, and manual = 0.94. This function is largely memory-bound and involves only simple arithmetic. As is common for reduction operations, loop collapsing resulted in worse performance than the trivial approach. When using *collapse*, the compiler may generate additional partial sums, which increases the frequency of updates to the reduction variable and adds overhead. If the chunk size is too small, threads may be assigned non-contiguous memory segments, reducing locality and harming performance. Unlike *init*, *norm* is not compute-bound and does not benefit from loop collapsing.

The *dxdt* function has higher arithmetic intensity than the other functions. All evaluated parallelisation strategies achieved notable speedups, although scalability varied with the number of cores. Because the interior points do not depend on the boundary points, the boundary updates can be executed in parallel without forcing the interior computations to wait. Using a *nowait* clause on the boundary loops allows interior calculations to overlap with boundary updates, reducing synchronization overhead and minimizing idle threads. Collapsing interior loops often degraded performance but was advantageous at select high core counts (e.g., 36) due to improved load balancing. The best speedups occurred for trivial interior-point parallelisation combined with boundary parallelisation using a *nowait* clause, yielding $f = 0.48$. The speedup and efficiency across different core counts for the final source file are presented in Table I.

TABLE I: Strong scaling results for the final program, parenthesis indicate error

	Number of cores					
Metric	2	4	8	16	32	64
Speedup	1.10(3)	1.883(2)	1.99(1)	2.372(6)	1.5(1)	1.33(3)
Efficiency	0.55(1)	0.4707(5)	0.249(1)	0.1483(4)	0.047(4)	0.0208(4)

IV. Conclusion

Combining the best implementations of each function into a single program yielded significant speedups. The final parallelisation choices were: a *collapse(2) for* loop in *init*, no parallelisation for *step*, a trivial *for reduction* in *norm*, and for *dxdt*, trivial interior-point parallelisation combined with boundary updates using a *nowait* clause. The measured serial fraction ($f = 0.48$) matches that of *dxdt*, indicating that it is the program's bottleneck.

V. Image Appendix

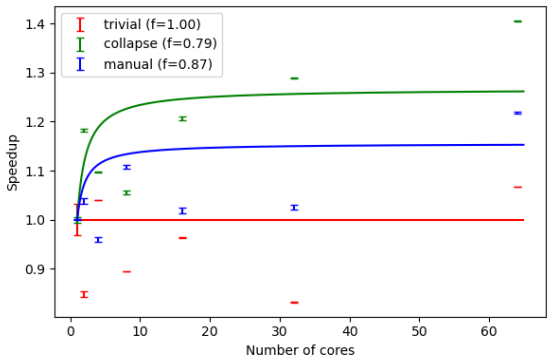

FIG. 1: Strong scaling performance of different parallelisation strategies for *init*.

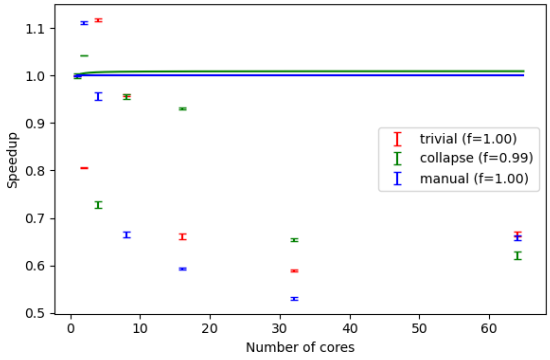

FIG. 2: Strong scaling performance of different parallelisation strategies for *step*.

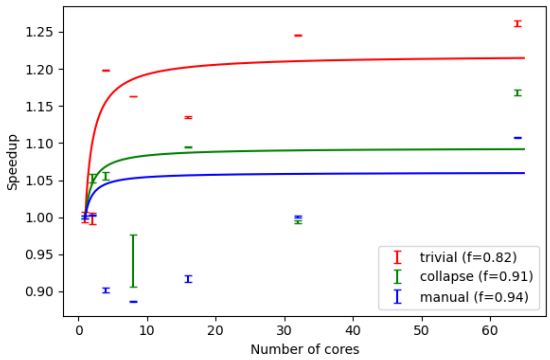

FIG. 3: Strong scaling performance of different parallelisation strategies for *norm*.

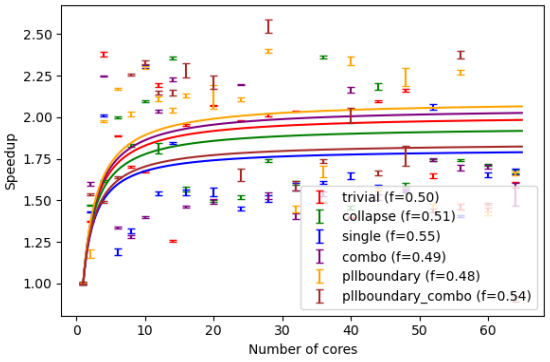

FIG. 4: Strong scaling performance of different parallelisation strategies for *dxdt*.

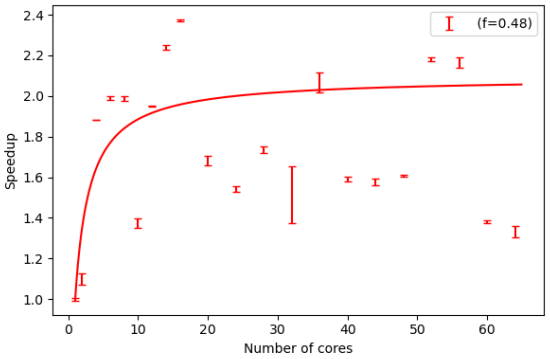

FIG. 5: Strong scaling performance of the final combined program.