Performance loss between concept and keyboard

András Z. Salamon[1,2] and Vashti Galpin[3]

[1] Computing Laboratory, University of Oxford
[2] Oxford-Man Institute of Quantitative Finance
[3] LFCS, School of Informatics, University of Edinburgh

Abstract. Standards bodies and commercial software vendors have defined parallel constructs to harness the parallelism in computations. Using the task graph model of parallel program execution, we show how common programming constructs that impose series-parallel task dependencies can lead to unbounded slowdown compared to the inherent parallelism in the algorithm. We describe various ways in which this slowdown can be avoided.

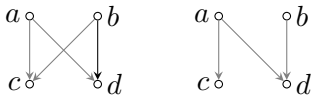
Inexpensive multicore processors have brought parallelism to the desktop computer [2] and users would like to take advantage of this parallelism for faster program execution. Standards for multiple-processor programming such as OpenCL [7] and commercial numerical software such as Matlab[4] and Mathematica[5] include language constructs for parallelism. Our position is that these constructs may limit the amount of parallelism, causing slowdown, but we also argue that there are ways to avoid this unnecessary loss in performance. With the projected progression from multicore computing (2-8 cores) to manycore computing (hundreds of cores) [10], we believe that parallel computing systems should avoid slowdown at the point of expressing the intention of the programmer, between the concept and the keyboard.

We focus on a specific structure on the dependencies between program tasks which some constructs impose. This structure is called series-parallel and can be most easily expressed as those task graphs generated by the language

$$P ::= seq(P, P) \mid par(P, P) \mid a$$

where a is a task or activity which represents some amount of program code (possibly as small as a single arithmetic operation) to be executed on one processor. Series-parallel task graphs are not only easy to express, but also have modular structure which can be exploited for efficient scheduling [3,11].

We can represent the tasks in a parallel program, and the dependencies between tasks, in a task graph (also known as an activity network). A task graph is a directed graph where each node is labelled with a distinct activity name, and associated with each activity is a positive real number, its duration, describing how long the activity will take to execute. Arcs in the task graph capture dependencies between tasks (also known as precedence constraints). The left graph in the figure is series-parallel and can be written as $seq(par(a, b), par(c, d))$. The right graph is not series-parallel and is the smallest graph lacking this property. It is called the N-graph. Denote by t the workload function that assigns durations to activities. The critical paths in a task graph determine the time to execute the whole graph. If $t(a) = t(d) = 1$ and $t(b) = t(c) = 2$, the series-parallel graph on the left will take 4 time units since the longest critical path is from b to c, whereas the N-graph on the right will take 3 time units to execute.



[4] Via the MATLAB Parallel Computing Toolbox. http://www.mathworks.com/
[5] From version 7. http://www.wolfram.co.uk/

Some dependencies in a task graph are inherent in the algorithm. Additional dependencies can be imposed by static scheduling, which maps the tasks to processors [6]. Dependencies can also be implicit in the programming constructs that are used to express the algorithm.

The left graph above is a series-parallelised version of the N-graph, obtained by adding an arc from activity b to activity c. (There are two other minimal ways to series-parallelise it.) By using the series-parallel (SP) graph with its extra dependencies instead of the non-series-parallel (NSP) N-graph, we can observe a slowdown of 4/3. This is in fact the least slowdown possible for this task graph and workload. Slowdown is defined as the ratio of the slower SP graph to the faster NSP graph. We can view the NSP graph as capturing the data dependencies in the original algorithm and the SP graph as the dependencies in a program expressing the algorithm in a specific programming language. In this specific case, the SP version will take one-third as long again to execute as the NSP version. The SP version can never be faster than the NSP version since dependencies are added. In some cases, this slowdown can be arbitrarily large.

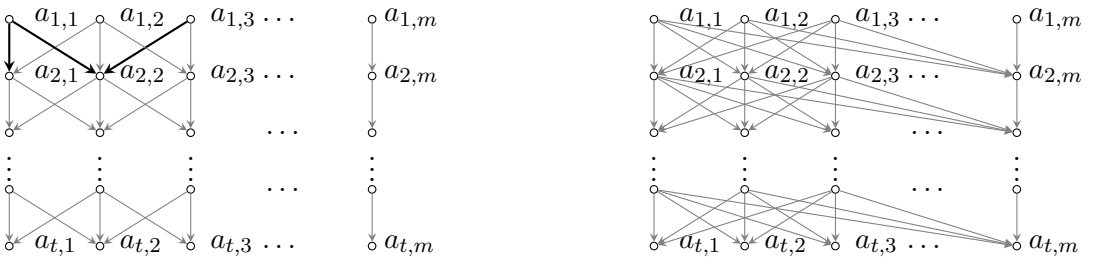
Now that we have introduced some concepts, we return to the programming constructs that can be used to exploit the parallelism provided by multi-core processors. Consider the example of a nested loop with neighbourhood synchronisation, which can be calculated sequentially using the Matlab style code

```
for i = 1 : t
  for j = 1 : m
    A[i,j] = f(A[i-1,j-1],A[i-1,j],A[i-1,j+1])        (task a_{i,j})
```

This can be expressed using a parallel-for construct as

```
for i = 1 : t
  parfor j = 1 : m
    A[i,j] = f(A[i-1,j-1],A[i-1,j],A[i-1,j+1])        (task a_{i,j})
```

which captures the idea that each of the new values in the array can independently be updated using the earlier values. If we study the dependencies inherent in the sequential code, we obtain the task graph on the left in the figure below. On the other hand, the code using the `parfor` construct imposes the dependencies shown in the right graph. Note that the left graph is not series-parallel as it contains the N-graph as indicated by the bold links (note the absence of the arc $(a_{1,3}, a_{2,1})$). The right graph can be expressed as $seq(par(a_{1,1}, \ldots, a_{1,m}), \ldots, par(a_{t,1}, \ldots, a_{t,m}))$ if we generalise par to take multiple arguments, and hence is SP.



In the NSP graph, there is no requirement that all tasks must finished before the second row is finished, so for example if $a_{1,1}, a_{1,2}, a_{1,3}$ have completed then $a_{2,2}$ can start regardless of

whether $a_{1,4}$ has finished. This is not the case in the SP graph which is a series-parallelisation of the NSP graph.

We now consider transforming an NSP graph to an SP graph. There are two classes of approaches to series-parallelising a task graph. In the first, workloads are not known, namely it is assumed that they are not known *a priori* [4]. In the second, workload information is used [9]. Note that in the case of the parallel-for construct, there is no explicit transformation as such but there is an NSP version given by the inherent data dependencies of the algorithm, hence this can be considered as a transformation without workload information.

For the case when workloads are not used in the series-parallelisation transformation or algorithm, it has been hypothesised that except for pathological workloads, the slowdown is bounded by two [12]. This can be expressed logically as

$$\exists Y \ \forall G \ \forall t \ T(G,t)/T(Y(G),t) \leq 2$$

where Y is an algorithm to convert an NSP graph to an SP graph that does not consider t, and $T(G,t)$ is the shortest possible time to execute G with workload t (obtained from the critical path of G with the largest execution time). Less formally, the hypothesis is that any task graph can be series-parallelised with slowdown at most 2, even if the details of the workload in obtaining the SP version are ignored.

In our opinion, this statement is false. We have shown that for every algorithm, it is possible to find a graph and a workload such that the slowdown is greater than two [9], and moreover these are not pathological cases. Specifically, for a neighbourhood synchronisation problem with depth 3, width 8 and degree 3 (the same degree as in the example above), where certain tasks have duration $4+\epsilon$ and the remainder duration 1, then slowdown is at least $(12+3\epsilon)/(6+\epsilon)$. This exceeds 2 whenever $\epsilon > 0$. In this case, the ratio of the longest task to the shortest task is just over 4, which is entirely feasible.

The proof of [9, Theorem 2] can be extended to larger task graphs to demonstrate arbitrarily large slowdown, as long as the choice of how to series-parallelise the task graph must be made without knowledge of the workload.

The above example of neighbourhood synchronisation has illustrated how the use of an inherently SP parallel-for construct can cause a slowdown of more than 2. This is clearly not desirable – this is a loss that occurs even before the system with parallel processors has had the opportunity to begin running the program. Next we move on to considering the options for dealing with this.

General techniques: One option is to choose a parallel language with more expressive constructs (ensuring that it does not lose parallelism unnecessarily during compilation). The disadvantage is that programming then becomes more complex and introduces the possibility of unexpected behaviour such as deadlocks.

Alternatively, it would be possible for numerical software such as MATLAB to introduce a parallelisation phase in which sequential code is assessed for data dependencies and these could then be used to introduce as much parallelism as possible [1,8].

Limited knowledge of workload: If it is known that the variance in the duration of tasks is neglible, then using inherently SP constructs is unlikely to lead to large slowdown. For a simple technique for series-parallelisation that gives level-constrained (LC) graphs, the slowdown is bounded by the ratio of the longest task to the shortest task [9]. If the variance is low, this will be close to 1, hence very little slowdown.

Full knowledge of workload: If workloads can be fully assigned to tasks in advance of execution, then careful hand-coding is a good approach, especially when the resulting program will be run frequently. Another approach, if an SP task graph is required before execution, is to consider the workload when transforming to the SP graph. We hypothesise that for algorithms which consider t that slowdown is bounded. Expressed logically,

$$\exists X \ \forall G \ \forall t \ T(G,t)/T(X(G,t),t) \leq 4/3$$

where X is a series-parallelisation algorithm which does use t to find the SP graph. This has been shown true for graphs with 4, 5 and 6 nodes. The proof for 6 nodes is programmatic and may be extendible to 7 and more nodes. However, it appears that finding the best SP version of a task graph may be NP-hard [9].

Note that the 4/3 bound is the best possible, illustrated by the N-graph with workload 1/2/2/1, as discussed previously. In a recent paper, the authors of the 2-bound hypothesis have shown experimentally with real workload data that slowdown of between 1 and 1.1 (and certainly less than 4/3) can be achieved [5].

In conclusion, unacceptable slowdown may be introduced by expressing an algorithm in series-parallel form. This can be counteracted if workload information is available, if more expressive programming constructs are used, or if the identification of parallelism is at least partially left to the programming environment. Finally, if the tasks in a program can be chosen to be of similar durations, then very simple techniques may be enough to avoid the slowdown discussed here.

References

1. A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pp. 221–235. Springer-Verlag, 1988. doi:10.1007/3-540-19027-9_15.
2. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, **52**(10), pp. 56–67, 2009. doi:10.1145/1562764.1562783.
3. L. Finta, Z. Liu, I. Mills, and E. Bampis. Scheduling UET-UCT series-parallel graphs on two processors. *Theoretical Computer Science*, **162**(2), pp. 323–340, 1996. doi:10.1016/0304-3975(96)00035-7.
4. A. González Escribano, V. Cardeñoso Payo, and A. J. C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proceedings, 1st Euro-PDS International Conference on Parallel and Distributed Systems*, pp. 251–256, 1997.
5. A. González-Escribano, A. J. van Gemund, and V. Cardeñoso-Payo. Performance implications of synchronization structure in parallel programming. *Parallel Computing*, **35**(8-9), pp. 455–474, 2009. doi:10.1016/j.parco.2009.07.002.
6. Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, **31**, pp. 406–471, 1999. doi:http://doi.acm.org/10.1145/344588.344618.
7. A. Munshi (editor). *The OpenCL Specification*, Feb. 2009.
8. S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multicores. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pp. 263–273. ACM, 2007. doi:10.1145/1274971.1275008.
9. A. Z. Salamon and V. Galpin. *Bounds on series-parallel slowdown*, Apr. 2009.
10. J. Shalf. The new landscape of parallel computer architecture. *Journal of Physics: Conference Series*, **78**(1), p. 012066, 2007. doi:10.1088/1742-6596/78/1/012066.
11. K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM*, **29**(3), pp. 623–641, 1982. doi:10.1145/322326.322328.
12. A. J. C. van Gemund. The importance of synchronization structure in parallel program optimization. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pp. 164–171. ACM, 1997. doi:http://doi.acm.org/10.1145/263580.263625.