18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

Multi-threading and performance tuning a hydrologic model: a case study

Perraud J.-M., Vleeshouwer J., Stenson M., Bridgart R.J.

Commonwealth Scientific and Industrial Research Organisation, Land and Water
Email: jean-michel.perraud@csiro.au

Abstract: The computer industry has evolved from single-core to many-core architectures to keep offering increasing processing power. Parallel programming is not new but in practice much remains to do to take advantage of these many-core architectures. Most software is still designed for serial execution, and the level of parallel execution is often limited to running the user interface and engine in separate threads to keep an application responsive. This paper focuses on task parallelization in an environmental modelling framework, The Invisible Modelling Environmental framework (TIME). The case study arises from a project requiring the calibration of rainfall-runoff models in numerous unimpaired gauged catchments located in Northern Australia. The hydrologic model structure is spatially distributed, such that each catchment model can have hundreds of input time series at a daily time step. These models are run on multi-core computers in a cluster, with the cumulated memory requirements possibly surpassing the available memory, slowing the computation to unacceptable levels due to virtual memory swapping. We thus tailor the number of catchment calibration tasks per compute node such that the cumulated memory footprint fits in the random access memory of that node. In order to still maximize the use of processing power on these multi-core nodes, we need to be able to parallelize these calibration tasks. Three parallelization strategies are considered, characterized mainly by different granularities for the tasks considered for parallelization: (1) parallelizing the calibration algorithm, (2) parallelizing the model along the spatial dimension, and (3) parallelizing the model along spatial and temporal dimensions. Assessing each against several criteria notably runtime performance gain, technological know-how, the amount of code changes and architectural impacts, solution (2) with multi-threading is preferred as the best compromise between these criteria. The architectural changes required by the parallelization and concomitant performance tuning are described along with the technical characteristics of .NET based solutions for multi-threading. We find that the performance tuning process necessary to make the parallelization a net benefit proves to be the bulk of the work. Two main performance bottlenecks are, not unexpectedly, identified: the use of software reflection (a.k.a. software introspection), and the access to input and output time series using date-time as an index. We use dynamic code generation to overcome the former, and introduce new interfaces for time series access for the latter, while avoiding pervasive changes to the framework. We find a satisfying speedup of roughly 80% of a theoretical linear speedup for a dual-threaded catchment model with 256 spatial grid cells. While additional threads improve the overall runtime, the runtime for eight threads is a somewhat disappointing 250% of the 'ideal' runtime, only partly explained by the expected parallelization overhead. Given the substantial architectural changes required in this case study that were not for parallelizing *per se*, we discuss the possible implications of the prevalence of multi-core processors on software design practices, at least in a scientific computing context.

Keywords: *Parallel programming, Multithreading, Performance tuning, TIME, HPC, Scientific computing*

1. INTRODUCTION

Parallel programming in different flavors (Central or Graphical Processing Unit, GRID, clusters, etc.) has recently gained visibility. Multi-core processors are now the norm for the majority of computers. However parallel programming remains an exception, and there is a large amount of software that has been designed implicitly for single processing units, including in environmental modelling.

This paper illustrates the process of parallelizing the calibration of a spatially distributed model to maximize the use of a cluster of nodes that have multiple cores. The modelling needs dealt with in this case study are representative of a large number of environmental modelling use cases, albeit not the most complicated ones. While this paper cannot be a generalized guidance for parallelizing modelling problem, several approaches are put forward and assessed for feasibility, providing elements for the reader to infer from. The reader will then find a more in-depth account of one technical solution using multi-threading.

2. CASE STUDY

2.1. Background

Between 2007 and 2009, CSIRO undertook a water resource assessment for the groundwater and surface waters of the Murray-Darling Basin (MDB), Northern Australia, South-West Western Australia and Tasmania. One component of the water yield assessment is the modelling of daily surface runoff as described in (Chiew et al. 2008). Runoff is modelled using $0.05^{o}\times0.05^{o}$ grid cells (~ 5 km×5 km), leading to c. 4×10^{4} grid cells over the MDB. When calibrating the catchment runoff models to historical flows, it is highly desirable to have all data in random access memory (RAM) to prevent any hard disk latency, in order to maximize the use of the processors. Given the hardware configuration of the cluster nodes, described thereafter, this proves feasible without much software modification for the unimpaired gauged catchments identified in South-Eastern Australia. In other words, it is possible to launch in parallel the calibration of one catchment per processor core without requiring virtual memory.

Northern Australia streamflow gauges define unimpaired catchments that are up to more than an order of magnitude larger in area than for the MDB. This leads to projected times of completion of the overall calibration process estimated to be of the order of a month, which is not acceptable. The runtime inflates dramatically because the large number of input time series cannot always fit in physical RAM. This occurs when several independent calibration tasks, each with a large memory footprint, are allocated concurrently to the cores of a same cluster node, i.e. the same physical machine. One way to prevent this is to allocate a number of “processors” (meaning here logical cores) of a cluster node for the tasks with large memory footprints, such that swapping to virtual memory on the hard drive does not occur. However this means that a single-threaded calibration task will run only on one of these allocated cores. We thus undertake the parallelization of catchment model structures, concomitant with a performance tuning exercise.

2.2. Hardware

While the parallelization and tuning process is not necessarily architecture specific, the hardware used for production model runs in this case study is a Windows Compute Cluster 2003 deployed on Windows Server 2003 64 bits. The cluster comprise up to twenty nodes, each with four physical processing cores. Hyperthreading is enabled, providing eight logical cores for each node. Each compute node has four Gigabytes of RAM. Development, performance benchmarking and some tests are also performed on dual- and single-core platforms.

2.3. Assessing the existing software

The software tools used for the modelling are largely command-line driven given the overwhelming need for batch running the modelling tasks. They are built upon TIME (Rahman et al., 2005), a modelling environment primarily used in Australia. Most modelling applications built on TIME have usually been single-threaded, barring the use of a background worker thread for the modelling engine to keep the user interface of the application responsive. The prevalence of lumped and parsimonious structures in many hydrologic models usually does not make a compelling case for multi-threading or multi-processing simulation models themselves. The main computationally intensive need for parallelism stems from model analysis techniques requiring multiple model runs such as calibration and Monte-Carlo based uncertainty assessments (Perraud et al., 2007, Davis et al., 2005). This need is at a high granularity and in practice often falls in the category of the (supposedly) “embarrassingly parallel” category, and is typically implemented using GRID computing software and/or cluster platforms.

Another characteristic of many applications built on TIME is the use of software reflection to decouple data and model structures, and provides a flexible mechanism for fostering model transparency (Perraud et al., 2005). The known downside is that software reflection is an expensive operation. It is *a priori* one "usual suspect" to watch for when reduction of simulation runtimes is required.

A third aspect of the existing software is that several subsystems, designed a decade or so ago, de facto assume a single model instance running in memory to perform a temporal or spatial simulation. In other words there is a built-in serial execution of multiple model runs. This is the case for most TIME-based modelling engines (model "runner") and the optimization/calibration tools. One rationale is often to avoid the creation of new model instances, indeed a performance boon in a non-parallel context. Most standard data structures are also not designed to handle concurrent access for the same reason. This has ramifications in terms of what is achievable to parallelize the calibration toolsets in this case study given a strictly limited time to implement the solution. Some of these ramifications will be pointed to in this paper when assessing the potential solutions.

3. ARCHITECTURAL CHANGES

3.1. Methodology

We follow some elements of the guidance in (Meier et al., 2004). The objectives scope the tests and benchmarks to put in place for change management. The main objectives of the exercise are to maximize the processor usage on each node and the speed of task completion. Network input-output and other such performance criteria are not an issue in the context of the case study. The repeatability of the results between single and multiple parallel tasks is highly desirable. This is not a given, as the sensitivity of the result to small rounding errors in an iterative parameter space search is very much possible.

The approach is to first define possible architectural changes to parallelize the process, without premature concern as to the initial lack of gains. Then we identify, diagnose and reduce the overheads in an iterative performance tuning process with the help of a performance profiler.

3.2. Possible approaches and feasibility

Given the modelling needs, three parallelization approaches with various granularities, not necessarily mutually exclusive, are considered and illustrated in Figure 1. Most calibration algorithms are in principle amenable to parallelization and the approach (1a) is a priori the most scalable across cores and cluster nodes using for instance a Message Passing Interface implementation (MPI) (Gregor and Lumsdaine, 2008). However using either MPI or multiple threads for (1a) clearly, and unfortunately, requires large changes to the code, something not feasible for the project time lines. This is compounded by the fact that the pre-existing calibration tool implementations are serial executors by design. On top of this it is worth mentioning that a technical aspect of MPI may make it difficult to use at least in approach (1a). Our understanding is that MPI is typically used for single program, multiple data (SPMD) problems. We are facing a single program, single data modelling need, although the exact classification may admittedly depend on the viewpoint. What is nevertheless clear is that the bulk input data is the same for each model simulation, and would need to be shared across parallel processes running on the same cluster node to avoid the memory issue we are trying to overcome. This is one further complication which would likely require significant changes or at least additions to the TIME data handling sub-systems.

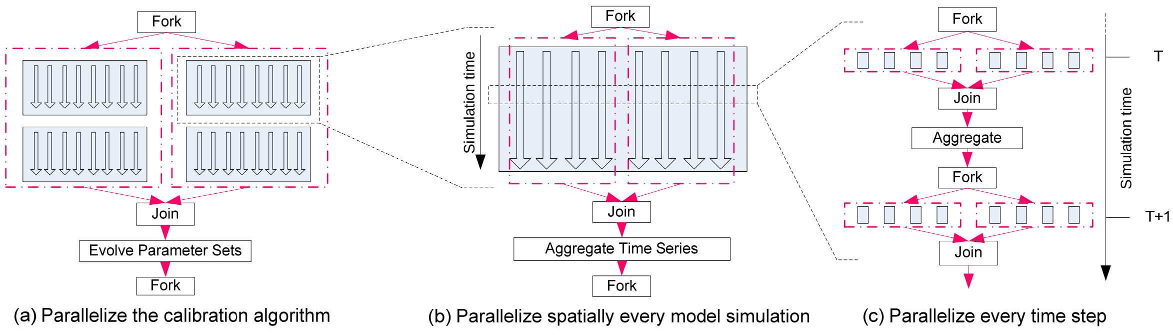

Figure 1. Three parallelization granularities illustrated for two parallel tasks. Grey boxes represent one catchment simulation run i.e. one iteration in the calibration algorithm. Each grey arrow represents one grid cell simulation run. Parallel threads are represented as magenta "dot-dash" boxes

The approach (1b) groups grid cell simulations in separate tasks. As there is no lateral flow between cells, no complex and costly tasks synchronization is required, a key criteria for feasibility. The required changes with multi-threading appear architecturally significant but manageable. Using MPI would however require more changes. Solution (1c) creates parallel calculations every time step and requires the smallest amount of code change, at least with multi-threading. However, and as expected, initial tests quickly confirm that the numerous fork/join operations have a clearly prohibitive cost, making it unviable. It should be noted that in modelling contexts where the model runtime at each time step is much larger than in this case study, (1c) may be a viable option. In this present case, (1b) is rated the most feasible.

3.3. Target Architecture

Given our limited know-how with MPI, and larger changes expected with it, the parallelization is undertaken with multi-threading. As a learning experience we use a preview release of the Microsoft Parallel Extensions Task Parallel Library (TPL) (Leijen and Hall, 2007), easily replaced later with the current production-level multi-threading tools in .NET. The core of the target architecture is presented in Figure 2. The change is transparent to the system outside of the model structure: the optimization tool (calibration algorithm) and its model runner are not impacted by the modification. This is mostly made possible by the pre-existing definition of `IGridCellYieldModels`, and the newly introduced `GridCellYieldModelFactory` and `ParallelGridCellYieldModels`. The determination of the number of tasks is a customizable property of the factory, as it is very unlikely an optimal number of parallel tasks can be determined in all circumstances.

In this instance the difference between TPL and the current .NET thread pool proves small, although the high-level call using the TPL `Parallel.For` method is indeed more elegant than the call to `ThreadPool.QueueUserWorkItem`. The higher level of abstraction of the TPL would be of a larger benefit in more complex multi-threaded contexts. Evolving the system to use the standard thread pool shows a runtime very similar to that with the TPL. The rest of this paper is based on performance measures taken with the use of thread pool.

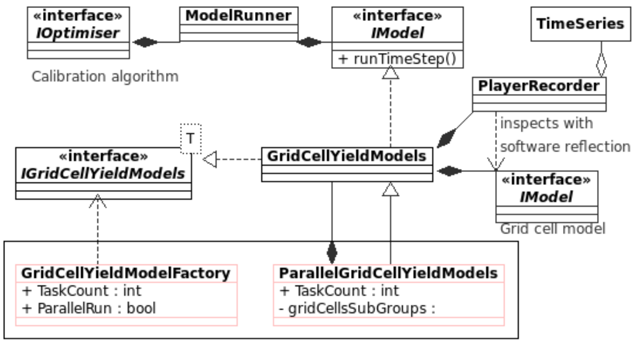

Figure 2. Multi-threading enabled, spatially gridded modelling structure

4. PERFORMANCE TUNING

4.1. Iterative performance tuning

A calibration test case is set up for a catchment with five grid cells. In order to evaluate the net performance loss induced by the architectural change, we run this test case on a single core processor with the baseline architecture and then with the new architecture enabled with two groups of cells (i.e. groups of resp. three and two cells), still on a single core processor, but running in series (single thread) instead of in parallel, in order to keep other things as equal as possible.

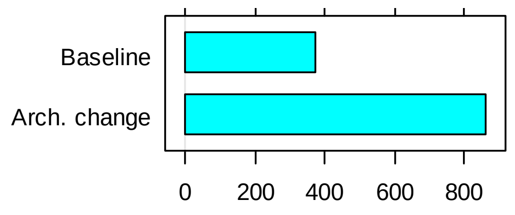

Figure 3: Runtime (s) after architectural changes

The introduction of the new architecture causes a significant loss of speed as shown in Figure 3. This is relatively unsurprising as the parallelization introduces additional buffering of time series that are populated using expensive operation such as software reflection.

Using a performance profiler, we confirm that the bulk of the runtime is at the level of running the model, to the tune of 80%. The overhead incurred by multi-threading, in itself, is minimal. The calibration tool itself represents a relatively small 10%, most of it spent logging information. There is no need to “optimize the optimizer”. Looking in more details it appears that more than half of the time is spent in handling setting input of, or getting outputs from, the model properties, of which almost 30% of it consists of software reflection operations as shown in Table 1.

Table 1. Software reflection hot spots

Function name	Inclusive %	Exclusive %
ModelRunner.setInputs	34.6	1.3
>>System.Reflection.RtFieldInfo.SetValue	19.8	0.3
ModelRunner.setOutputs	23	0.6
>>System.Reflection.RtFieldInfo.GetValue	9.4	0.2

To reduce this overhead, we introduce a modification of the TIME subsystems that “records” or “plays” time series, the main source of these software reflection hotspots. A “player” (resp. “recorder”) in TIME is in essence a dictionary where keys are `ReflectedAccessor` objects (Figure 4) and values are input (resp. output) time series. The `ReflectedAccessor` is a high-level tool to get or set arbitrary properties of arbitrary objects, using software reflection. Dynamical generation of delegates (i.e. methods) to set and get model properties is implemented as an option for `ReflectedAccessor`, replacing software reflection. It is based on the work of (Randson, 2007). The change is largely confined to the `ReflectedAccessor` class, and transparent to the `Player` and `Recorder` that use it. This change, along with other minor tuning, removes nearly 26% of the runtime as expected. In subsequent rounds of performance profiling, the main hot spots relate to the use of enumerators (lists and dictionaries with the `foreach` keyword) and the indexing of time series by `DateTime` objects as shown in Table 2.

Table 2. Time series indexing and collections iteration hot spots

Function name	Inclusive %	Exclusive %
TimeSeries.itemForTime(DateTime)	9	0.4
TimeSeriesStore.set_Item(DateTime, object)	7.5	0.2
Collections’ GetEnumerator and MoveNext	~10	NA

Enumeration costs are alleviated when possible by either altogether renouncing to using collections or caching their entries in arrays in private class members. The indexing of time series by `DateTime` objects is a flexible mechanism which confines the offsetting of array indexes within the `TimeSeries` class, rather than requiring a system orchestrating model executions (“model runner”) to handle a possibly complex set of index offsetting that may vary from one time series object to another. This flexibility comes at a cost that is too high in our calibration context. A set of interfaces for fast time series indexing is designed (Figure 4). The responsibility of offsetting remains largely with the time series, but the “model runner” now has the option to read or write the time series by indexing it with integers, as long as the time series implements `ITimeSeriesFastAccessorProvider`.

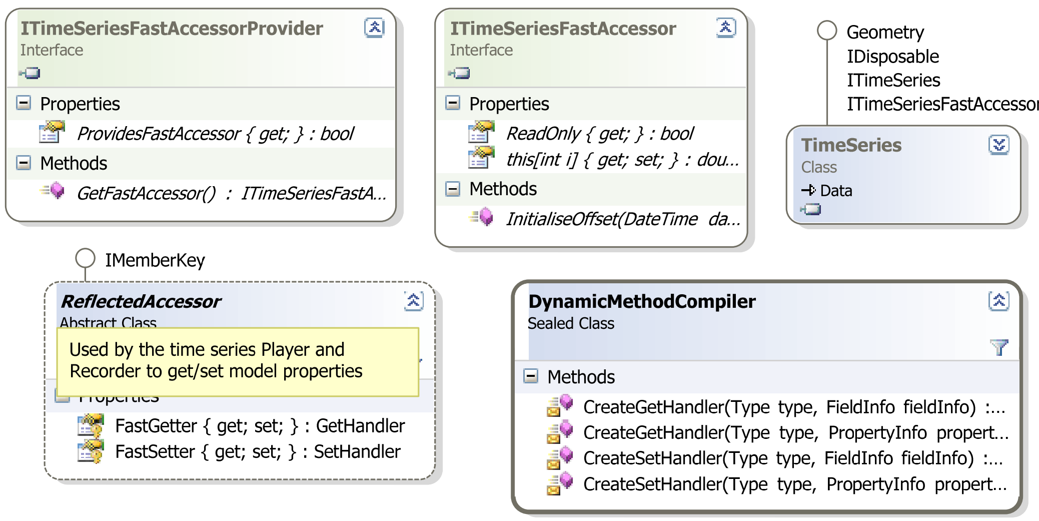
ITimeSeriesFastAccessorProvider
Interface
Properties
ProvidesFastAccessor { get; } : bool
Methods
GetFastAccessor() : ITimeSeriesFastA...
ITimeSeriesFastAccessor
Interface
Properties
ReadOnly { get; } : bool
this[int i] { get; set; } : dou...
Methods
InitialiseOffset(DateTime da...
Geometry
IDisposable
ITimeSeries
ITimeSeriesFastAccessorProvider
TimeSeries
Class
Data
IMemberKey
ReflectedAccessor
Abstract Class
Used by the time series Player and Recorder to get/set model properties
Properties
FastGetter { get; set; } : GetHandler
FastSetter { get; set; } : SetHandler
DynamicMethodCompiler
Sealed Class
Methods
CreateGetHandler(Type type, FieldInfo fieldInfo) :...
CreateGetHandler(Type type, PropertyInfo propert...
CreateSetHandler(Type type, FieldInfo fieldInfo) :...
CreateSetHandler(Type type, PropertyInfo propert...

Figure 4. Adding dynamic code generation and fast time series indexing

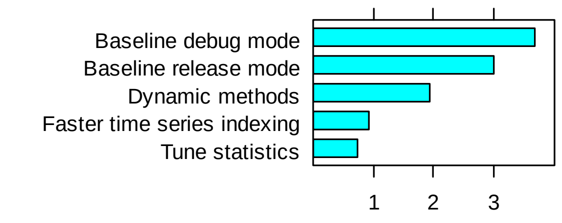

Figure 5. Runtimes (s) through tuning

Figure 5 shows the overall substantial improvement with the iterative remediation to performance hot-spots, using an application that executes a much smaller calibration process but with a high process priority to reduce the uncertainty of the measures. The gains are half an order of magnitude. However switching the test platform to a dual-core processor, the single-threaded calibration still outperforms the dual-threaded one: parallelizing for a catchment of only five cells is counterproductive.

4.2. Results

Figure 6 shows measured runtimes for a test catchment set up with 256 cells. There is as expected a speed-up up to the number of logical cores (8), but the relative difference to a linear speed-up is growing disappointingly larger with the increasing number of threads, even considering that the theoretical linear speedup could be achieved only by parallelizing all of the code. When running 8 threads the overall CPU use is ~80%, meaning that ~20% of the overall calibration task is still non-parallel, a number that was roughly expected. This alone cannot account for the difference of a factor 2.5 between the theoretical and measured runtimes. While the overall performance improvement exercise is clearly successful (the overall production runtime applied to Northern Australia catchments was reduced from an initially *projected* 40 days down to around 2 days), further investigation in this overhead increasing with the number of core is warranted. Designing an appropriate performance profiling with multi-threading would be more complex than what was done for this paper and will hopefully be tackled in the near future. It is also worth mentioning that we suspect that a sub-optimal data locality and the occurrence of on-chip memory cache misses is impacting performance, but is hard to measure. (Toub et al., 2008) offers a good introduction to this type of issues.

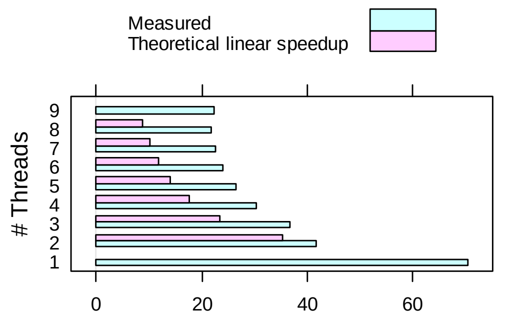

Figure 6. Runtime (min) for a 256 cells catchment

5. DISCUSSION

While the pivotal aspect in this paper is parallelizing a model structure, the design and implementation of the parallel architecture proves to require less effort than the subsequent performance tuning necessary to make this parallelization worthwhile. The performance hot-spots are also impacting the single-threaded baseline case as illustrated by Figure 5, but their effect is magnified by multi-threading the catchment model. This reflects the prevalent practice of not optimizing code for performance prematurely. Admittedly and with the benefit of hindsight the performance tuning could have beneficially been carried out even without the need for multi-threading.

Performance tuning means more code, hence more complexity and risk, which must be avoided unless necessary. That being said, the size of the architectural changes required to achieve the performance tuning is significant, and it begs the question whether performance should be included earlier in the design process as one criterion on par with others, as long as it is driven by real use cases and properly resourced from a project perspective. The practice of avoiding premature optimization may have been taken too far, leading to situation where it is dealt with only when it has become a blocking problem. Multi-core is now the norm, and designing parallel software is brought to the forefront. Toolsets such as parallel debugging tools, the TPL and the recently increasing interest in functional programming languages built on top of Java and .NET should make parallel programming more affordable and widely used. This may imply that performance issues will be brought to the fore earlier than they have been previously.

6. CONCLUSIONS

The case study of calibration of a gridded spatio-temporal runoff model is found suitable for parallelization at several granularities, using MPI or multi-threading. Parallelizing the spatial structure using threads is assessed as the most feasible solution given the constraints: needs, timelines, know-how and target hardware. A modified model architecture with a confined impact is presented, using in the design process a preview of a parallelization toolbox, the Task Parallel Library, thereafter replaced by the current production-level thread classes. The architectural changes for parallelization incur an overall prohibitive overhead cost without additional performance tuning. An iterative performance tuning process is presented, and proves the bulk of the work required to make the parallelization a benefit. The main performance issues are the use of software reflection and indexing of time series data by date and time. We use dynamic code generation and add time series interface definition to overcome these bottlenecks without compromising the existing features and behaviors. We find that dual-threading a model comprising 256 cells executes in 1.18 times the theoretical runtime, assuming a linear speedup. The overhead increases with the number of threads to a relatively disappointing factor of 2.5 for eight threads. We suggest that multi-core platforms will require some changes in the mindset and practices of modellers and programmers. New parallel toolboxes and functional languages should make this more affordable but will not negate the need to consider hardware more, perhaps something that the popularity of virtual machines over the past decade has eclipsed.

ACKNOWLEDGMENTS

This work was undertaken as part of the Northern Australia Sustainable Yields project and the National Water Commission Catchment Water Yield Estimation Tool project. We gratefully acknowledge Ian Szarka, Peter Fitch and two anonymous reviewers for their very helpful comments and suggestions.

REFERENCES

Chiew F.H.S., J. Vaze, N.R. Viney, P.W. Jordan, J.-M. Perraud, L. Zhang, J. Teng, W.J. Young, J. Penaarancibia, R.A. Morden, A. Freebairn, J. Austin, P.I. Hill, C.R. Wiesenfeld and R. Murphy (2008). Rainfall-runoff modelling across the Murray-Darling Basin. Water for a Healthy Country Flagship. CSIRO. 70 pp.

Davis G., R. Bridgart, T. Stephenson and J. Rahman (2005). Adding Grid Computing Capabilities to an Existing Modelling Framework. In Zerger, A. and Argent, R.M. (eds) MODSIM 2005 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2005, pp. 690-696. ISBN: 0-9758400-2-9

Gregor D. and A. Lumsdaine (2008) Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, p. 133—142, 20-23 February 2008.

Leijen D. and J. Hall (2007), Optimize Managed Code For Multi-Core Machines, MSDN Magazine, October 2007

Meier J.D., S. Vasireddy, A. Babbar and A. Mackman (2004), Improving .NET Application Performance and Scalability, patterns & practices, Microsoft Corporation, ISBN 0-7356-1851-8

Perraud, J.-M. , S. P. Seaton, J. M. Rahman, G. P. Davis, R. M. Argent and G. D. Podger (2005) The architecture of the E2 catchment modelling framework. In Zerger, A. and Argent, R.M. (eds) MODSIM 2005 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2005, pp. 690-696. ISBN: 0-9758400-2-9

Perraud, J.-M., Kuczera, G. & Bridgart, R.J. (2007), Towards a Software Architecture to Facilitate Multiple Runs of Simulation Models. In Oxley, L. and Kulasiri, D. (eds) MODSIM 2007 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2007, pp. 846-852. ISBN: 978-0-9758400-4-7.

Rahman, J.M., J.-M. Perraud, S.P. Seaton, H. Hotham, N. Murray, B. Leighton, A. Freebairn, G. Davis & R. Bridgart (2005), Evolution of TIME. In Zerger, A. and Argent, R.M. (eds) MODSIM 2005 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2005, pp. 697-703. ISBN: 0-9758400-2-9.

Randson H. (2007), Dynamic Code Generation versus Reflection, The Code Project, last accessed 2009-03-21, http://www.codeproject.com/KB/cs/Dynamic_Code_Generation.aspx.

Toub S., I. Ostrovsky and H. Yildiz (2008), False Sharing, MSDN Magazine, October 2008