Chapter 3: Computing in Education

Parallel and Distributed Computing in Education (Invited Talk)

Peter H. Welch

Computing Laboratory,
University of Kent at Canterbury, CT2 7NF, United Kingdom
P.H.Welch@ukc.ac.uk

Abstract. The natural world is certainly not organised through a central thread of control. Things happen as the result of the actions and interactions of unimaginably large numbers of independent agents, operating at all levels of scale from nuclear to astronomic. Computer systems aiming to be of real use in this real world need to model, at the appropriate level of abstraction, that part of it for which it is to be of service. If that modelling can reflect the natural concurrency in the system, it ought to be much simpler.

Yet, traditionally, concurrent programming is considered to be an advanced and difficult topic – certainly much harder than serial computing which, therefore, needs to be mastered first. But this tradition is wrong.

This talk presents an intuitive, sound and practical model of parallel computing that can be mastered by undergraduate students in the first year of a computing (major) degree. It is based upon Hoare’s mathematical theory of Communicating Sequential Processes (CSP), but does not require mathematical maturity from the students – that maturity is pre-engineered in the model. Fluency can be quickly developed in both message-passing and shared-memory concurrency, whilst learning to cope with key issues such as race hazards, deadlock, livelock, process starvation and the efficient use of resources. Practical work can be hosted on commodity PCs or UNIX workstations using either Java or the Occam multiprocessing language. Armed with this maturity, students are well-prepared for coping with real problems on real parallel architectures that have, possibly, less robust mathematical foundations.

1 Introduction

At Kent, we have been teaching parallel computing at the undergraduate level for the past ten years. Originally, this was presented to first-year students before they became too set in the ways of serial logic. When this course was expanded into a full unit (about 30 hours of teaching), timetable pressure moved it into...
the second year. Either way, the material is easy to absorb and, after only a few (around 5) hours of teaching, students have no difficulty in grappling with the interactions of 25 (say) threads of control, appreciating and eliminating race hazards and deadlock.

Parallel computing is still an immature discipline with many conflicting cultures. Our approach to educating people into successful exploitation of parallel mechanisms is based upon focusing on parallelism as a powerful tool for simplifying the description of systems, rather than simply as a means for improving their performance. We never start with an existing serial algorithm and say: ‘OK, let’s parallelise that!’ And we work solely with a model of concurrency that has a semantics that is compositional – a fancy word for WYSIWYG – since, without that property, combinatorial explosions of complexity always get us as soon as we step away from simple examples. In our view, this rules out low-level concurrency mechanisms, such as spin-locks, mutexes and semaphores, as well as some of the higher-level ones (like monitors).

Communicating Sequential Processes (CSP)[1,2,3] is a mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects. Developed by Tony Hoare in the light of earlier work on monitors, CSP has a compositional semantics that greatly simplifies the design and engineering of such systems – so much so, that parallel design often becomes easier to manage than its serial counterpart. CSP primitives have also proven to be extremely lightweight, with overheads in the order of a few hundred nanoseconds for channel synchronisation (including context-switch) on current microprocessors[4,5].

Recently, the CSP model has been introduced into the Java programming language [6,7,8,9,10]. Implemented as a library of packages [11,12], JavaPP[10] enables multithreaded systems to be designed, implemented and reasoned about entirely in terms of CSP synchronisation primitives (channels, events, etc.) and constructors (parallel, choice, etc.). This allows 20 years of theory, design patterns (with formally proven good properties – such as the absence of race hazards, deadlock, livelock and thread starvation), tools supporting those design patterns, education and experience to be deployed in support of Java-based multithreaded applications.

2 Processes, Channels and Message Passing

This section describes a simple and structured multiprocessing model derived from CSP. It is easy to teach and can describe arbitrarily complex systems. No formal mathematics need be presented – we rely on an intuitive understanding of how the world works.

2.1 Processes

A process is a component that encapsulates some data structures and algorithms for manipulating that data. Both its data and algorithms are private. The outside