University of Southampton

Deadlock free algorithmic Parallelism

Analysis, Implementation and Performance

D.P.Simpson

Doctory of Philosophy

Department of Electronics and Computer Science

May 2003

University of Southampton

ABSTRACT

Faculty of Enginering
Depertment of Electronics and Computer Science

Doctor of Philosophy

Deadlock Free algorithic parallelism: Analysis, Implementations and Performance

by D.P.Simpson

Networks of cheap, readily available processors provide an increasingly common solution to the ever-increasing demand for computational resources. Massively parallel computers are now in common use in a wide variety of applications, including many areas of signal and image processing, automotive and aerospace engineering, and scientific research. The concurrent development of suitable programming environments is essential to the efficient exploitation of these parallel computational architectures. This thesis addresses this problem, and describes the design, implementation, and utilization of a programming library for the efficient exploitation of computational parallelism on homogenous and heterogenous workstation clusters.

The analysis of computational parallelism is complicated by a number of factors. The state space is increased, parallel programs may exhibit non-deterministic characteristics, and it is frequently impossible to decompose an algorithm executed in parallel into simple, composable logic elements. This may lead to erratic, unpredictable, and irreproducible behaviour in the execution of such programs because any errors incorporated into a code my be dependent on the precise sequence in which they are executed. Formal mathematical analysis is, however, sufficiently powerful to prove that a program is correctly structured, and is a valuable approach in establishing the expected behaviour of a program where it is too complex to be accessible to inspection alone. The use of formal mathematical methods in the analysis of computational parallelism plays a central role in this thesis.

It is proven in the first section of this thesis that a simplified communication scheme, defined as the MP parallel programming language[30], is deadlock free. This key result is established using the process calculus CSP[15], providing a precise description of the components which would otherwise be liable to subtle ambiguities of interpretation if presented in seemingly 'plain English'. In particular, it is shown that the CSP design adopted here guarantees freedom from deadlock, provided that all input and output operations are performed simultaneously and an infinitely readable 'external input' is provided.

A library which closely follows the CSP design has been implemented and is presented in detail. This presentation describes both the construction of the library and the syntax required for its use. The correctness of the implementation is then demonstrated using illustrative examples. An exhaustive analysis of a common sorting algorithm is presented which is sufficiently simple that the flow of data in the program may be traced in detail. The second, more complicated example is drawn from an important problem commonly occuring in atomic, molecular and nuclear physics, and which is a significant consumer of computational resources worldwide; the Hartree-Fock approximation. The atomic Hartree-Fock method is investigated using the `mpkern` parallel programming library in both task parallel and data parallel forms. An analysis is also performed which examines the efficient distribution of the most computationally intensive tasks in the Hartree-Fock method across a heterogeneous network of nodes, achieving close-to-linear scaling in the cases tested. The performance and potential uses of the library are discussed in the concluding section of the thesis.

Contents

List of Figures

List of Tables

Acknowledgements

The author wishes to thank his supervisor Dr J.S. Reeve for his support during the many twists and turns of this work and for suggesting the approach used in developing the new parallel programming library. He wishes to express his thanks to Dr H. Quiney (Melbourne) for suggesting that the new library be applied to the solution of the atomic Hartree Fock equations and for reading the manuscript of this thesis. Dr. H. Quiney also assisted with the mathematical description of the Hartree-Fock procedure and provided a sequential fortran 77 implementation on which the versions discussed here are based (which are numerically identical to the original fortran version). He is most grateful to Dr. J.W. Sanders (Oxford) and Prof. A.W. Roscoe (Oxford) for helpful discussions on deadlock. The deadlock freedom proofs are based on an argument communicated to the author by Prof. A.W. Roscoe. He wishes to thank the EPSRC for a grant and his parents for their support.

Chapter 1

Introduction

Parallelism is often used to make feasible the implementation of computationally intensive tasks which would otherwise be beyond reach. Scientific and technological applications of large-scale parallel computing are now frequently used to construct realistic physical models or to perform simulations of complex phenomena which may evolve in time. Notable areas of application to physical processes include computational fluid dynamics, in particular meteorology and aerodynamic simulations, financial modelling, the simulation of solid- and liquid-state properties in materials science, and the electronic structure and chemical reactivity of atoms and molecules in many areas of chemistry, physics and molecular biology.

The increasing the speed of processors in the last 10 years has made many of the problems that used to require parallel systems feasible on a single processor. There are still applications, for example weather forecasting, for which the speed of current single processor systems is insufficient. Large scale numerical problems also arise in computational fluid dynamics, quantum physics and chemistry, finite element analysis and image processing. Many of these applications can exploit parallel processing, which is the simultaneous use of multiple processors to attack a problem.

Unfortunately the use of parallel processing introduces the possibility of race conditions and deadlock, which only affect concurrent systems. The complexity of analysis is also significantly increased by parallel processing.

Race conditions occur when one processor manipulates shared data that is in an inconsistent state because another processor is in the process of updating it. Examples include a processor reading the value of a shared number before another processor has updated it.

Deadlock occurs when no processor can make progress because they are all waiting for another processor to do something. This situation frequently occurs because all the processors want to send data to a processor that is unwilling to receive it. Deadlock can also occur when a race condition causes an inaccurate value of a counter which indicates the number of processors that have reached a synchronisation point.

Testing a parallel program is difficult because bugs may be exposed only by specific timing, which is almost impossible to reproduce reliably. Thus there is no guarantee that a program that passes a test will always operate correctly. The only alternative is to locate and eliminate bugs by analysis of the program.

Sequential programs are frequently analysed by making statements about their state at particular points. Analogous statements about parallel programs using p processors must take into account the code being executed by all p processors at a given moment, which may result in an

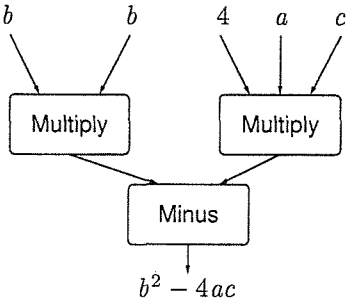


Figure 1.1: Task parallel network of components that computes $b^2 - 4ac$

exponential growth in the number of states that have to be examined. This makes global analysis of any significant program impractical. The need to take account of mutual exclusion devices, used to prevent race conditions, increases the complexity of global analysis.

Various design restrictions have been used to reduce the complexity of the analysis required to verify a parallel program, for example algorithmic parallelism. Algorithmic parallelism, also known as task parallelism, avoids race conditions by eliminating shared data. The program is broken up into a series of components which transform their inputs into outputs which are sent to appropriate destinations. Composable logic applies to task parallel programs: the overall effect of several components can be simply deduced from the processing performed by each component. It is usually assumed that a message can only be sent if the destination is willing to receive it.

Figure 1.1 illustrates a task parallel method of computing $b^2 - 4ac$. The arrows represent the direction of data flow, the multiply components compute a product of their inputs and the minus component subtracts its right input from its left input. If sufficient resources are available then both the multiply components can operate in parallel. In practice the example in figure 1.1 would be much slower than a sequential solution on most hardware, due to the cost of communication. The same design can be applied to larger problems using computationally expensive components.

Unfortunately task parallel programs can still deadlock for several reasons. The usual example is a cycle of components all trying to send a message. All the components in such a cycle are blocked because the intended recipient is not willing to receive a message until it has sent its message, and thus no further progress is possible. Similar situations can arise due to aggregating several components into a single thread of control, which may be necessary for an efficient implementation of a task parallel program.

This thesis introduces two styles of task parallel programming which merge the input and output stages into a single parallel input and output stage. This eliminates many of the problems of cycles because all components that wish to send an input must be willing to accept one. Two types of systems are analysed; synchronous programs and asynchronous programs.

Synchronous programs are arbitrary networks of synchronous components, which read an input from all their inputs and send an output via all their outputs before entering their compute phase. All inputs are readable before the first cycle. Deadlock is impossible for such a system and a property relating the number of cycles completed by two indirectly connected components can be proved.

Asynchronous programs, which are networks of asynchronous components, are much more flexible than synchronous programs. Asynchronous components can maintain state and choose which, if any, of their outputs to use. However an asynchronous component must be willing to accept one value from *any* input that has been used. If an asynchronous component is unable to

enter its compute phase due to an unfinished output it might be required to read a single value from more than one input. An asynchronous component must enter its compute phases when all its outputs, generated in the previous cycle, have completed and one or more input has been read.

This thesis presents a proof that *any* network of asynchronous components is deadlock free, provided that at least one infinitely readable external input is available and the computation phase of all the components always terminates. The "rules" used to determine which outputs are used and how the components are connected do not matter. The communication network connecting the components can contain arbitrary cycles. Attractions of the design include

- the absence of restrictions on the way the components are connected.
- the ability to exercise none of a component's outputs.
- lack of global synchronisation.
- guaranteed deadlock freedom without *any* analysis of the way the components are connected.

A library implementing the asynchronous design, without the external inputs and designed for a batch processing environment, is presented. Benchmark results of a real application indicate the library is efficient and the design should be scalable. The library supports components with (private) state, which can be different for each component even if they share the computation phase implementation. Termination is implemented by a mechanism unrelated to the task parallel program design. The only possible reason for deadlock is that no component can read any of its inputs and this is easy to avoid. For example, it is easy to see that the minus component in figure 1.1 will receive both the inputs it requires.

1.1 Outline of this document

Chapter 3 briefly describes CSP, the process calculus used to analyse the behaviour of synchronous and asynchronous programs. Terminology such as the precise meaning of "deadlock free" and a few simple results are proved in this chapter for later use.

Chapter 4 shows that synchronous programs, where every component is obliged to send a message to every output and accept a communication from every input in every cycle, are deadlock free. A brief analysis of the strength of synchronisation in a synchronous program is also included. Chapter 5 proves the more surprising result that asynchronous programs, which only require components to be willing to read a value from one or more inputs per cycle, are also deadlock free. Both deadlock freedom proofs postulate a trace after which deadlock occurs and deduce contradictions about it.

FDR, a CSP based verification tool, can only be used to analyse specific instances and thus was not appropriate. FDR was used to show that a 5 node complete graph (5 components with a direct, bi-directional connection between all pairs of components) is deadlock free. This result is powerful because it also applies to any 5 node program that uses a subset of the connections and does not depend on the rules that determine which connections are used.

Since even a 5 node complete graph has many thousands of states a large and complex network, for example the task parallel Hartree-Fock program below, could have too many states to analyse using FDR or a similar tool. The proof eliminates this problem because it analyses a network of components without reference to the number of components or how they are connected.

Chapter 6 describes how to use a library for implementing programs which are almost identical to asynchronous programs. The timing guarantees, high level details of the implementation and programming interface are described. Methods for dealing with pathological timing, and the functions included in the library to simplify this task, are documented.

A simple programming language, called cg, for generating the required list of components and connections between them is described. An implementation of sample sort, a popular parallel sorting algorithm, is used as a simple example.

Chapter 7 describes the implementation of the library. The implementation of cg is also described briefly. The claim that the library is a correct implementation of the model is a strong claim because the library is single threaded and the programming model uses many threads. It is obviously important to show that the use of the library cannot introduce deadlock; the two examples in chapter 2 show that this can happen if multiple, separate, components are merged into a single thread.

The description of the library internals includes sufficient detail to discuss why the library implementation is a correct implementation of the described model. A complete proof of this assertion is outside the scope of this thesis. Among other things, a complete proof would have to demonstrate the maintenance of several invariants involving variables omitted from chapter 7. Chapter 8 describes some of the missing details and argues that some of the most important invariants are maintained.

Chapter 9 applies the library to a simple version of a "real world" large scale numerical problem, the atomic, closed shell, Hartree-Fock problem. Large instances of the molecular Hartree-Fock problem are often solved on supercomputers. The implementation of a task parallel and data parallel version of the problem, both using the `mpkern` library, is briefly described and the performance of the resulting programs measured.

Finally chapter 10 draws some conclusions and indicates some of the possible areas for further work.

Chapter 2

Parallel programming models

Parallel programming involves the solution of a wide range of problems on a wide range of systems. The cost of accessing remote memory, where it is supported, depends on how the shared memory is implemented. Accessing globally flat shared memory costs the same on all processors and is usually inexpensive. Large scale shared memory supercomputers often have distributed shared memory with non-uniform access times which depend on where the memory is located—access to "local" memory is much faster than access to "remote" memory.

An increasing popular design is a cluster: multiple networked commodity computers, which are relatively cheap due to the economies of scale, used only for parallel computation tasks. The network that connects the nodes of a cluster is usually a lot slower, and cheaper, than the networks used by supercomputers. The nodes on a cluster do not share any memory. Message passing is usually used on clusters, but software simulation of shared memory is possible (albeit expensive).

This diversity of hardware has lead to the development of various models of parallel programming that can be implemented on a range of parallel systems. The performance of these models on any given hardware can usually be described by a relatively small set of statistics. For example vector machines are described by parameters including the throughput when all the stages of the pipelined vector processors are in use, and the vector length which is required to achieve half this performance.

Source compatible implementations of some models are available on a wide range of systems, allowing a program that is used to solve large instances of a problem on a supercomputer to be developed and tested, using small examples, on relatively inexpensive and widely available hardware. The program can then be recompiled, without any changes, for use on a supercomputer.

It is worth examining a problem before deciding to use parallel computation. The improvement possible on some problems is small and it may be cheaper, and easier, to use a faster computer instead. Amdahl thought the impact of parallel computing would be relatively modest due to the serial portions of jobs which cannot be done more quickly on multiple processors.

2.1 Amdahl's Law

Problems that can be decomposed into completely independent elements, for example, rendering many frames in a film, scale linearly—p processors do the job p times faster. However many problems have portions which can be done in parallel and portions which must be done serially.

If $T_p(n)$ is the time taken to solve an instance of size n using p processors, $T_s(n)$ is the portion

that must be done serially and $T_c(n)$ is the time taken for the portion that can be parallelised then

$T_1(n) = T_s(n) + T_c(n)$. The best performance that can be achieved using p processors is $T_s(n) + T_c(n)/p$ and thus

$$\frac{T_1(n)}{T_p(n)} \leq \frac{T_s(n) + T_c(n)}{T_s(n) + Tc(n)/p} < \frac{T_1(n)}{T_s(n)}$$

This is known as Amdahl's law and dates back to 1967. Problems that scale linearly, or almost linearly, mentioned above, have a negligible serial time, $T_s(n)$, for all n. Parallelisation, and communication, often has significant costs so very few programs achieve the speed up permitted by Amdahl's law.

Fortunately large numerical problems often have a small $T_s(n)/T_c(n)$ and $T_c(n)/T_s(n) \to 0$ as $n \to \infty$, so if parallelism is used to solve larger cases the impact of Amdahl's law is significantly reduced. Choosing scalable algorithms, which might be less efficient on a single processor, reduces the impact of the serial time in many cases.

2.2 Programming models

Different sorts of parallel systems and hardware have different characteristics. The wide variety of parallel systems has led to the development of a number of programming models that describe parallel systems. The challenges involved in parallel programming include:

- A parallel program has many more possible states than a sequential program. Analysing all the possible states, to ensure correctness, whether formally or informally, is likely to be intractable. Simplifying design restrictions can reduce this problem.
- Bugs in parallel programs can depend on very particular timing, which is not necessarily reproducible, even if it is known. Adding tracing might perturb the timing sufficiently to avoid the bug. System features, for example buffering, might prevent the bug from occurring in small test cases.
- Hardware can include components of various ages, and as a result nodes of a mixture of speeds. The time required to compute a complex expression may be a complex function of the processor type and speed, making dividing a task into equal cost portions difficult.
- Poor *load balancing* can cause inefficient use of resources due to different processors arriving at a synchronisation point at different times. Dynamic load balancing algorithms attempt to redistribute the load in the light of actual performance, at the cost of using processor time for an activity that does not advance the computation.

One of the significant differences between different parallel programs is the granularity. A fine grained program is broken into many small operations, which are combined frequently. A coarse grained program is broken into a few large jobs, which synchronise less frequently. Fine grained programs can, in some cases, exploit more parallelism but may be slower due to the cost of synchronisation. Communication on clusters is usually much slower than computation, so clusters are inappropriate for fine grained parallel programming.

2.2.1 Shared memory

If the number of processors is moderate it is feasible to construct parallel computers with globally shared memory, to which all processors have uniform access. This often referred to as globally flat memory. Multi-processor x86[1] servers are a widely available example of this sort of system. The hazards on these machines are *race conditions* and *deadlock*.

Many data structures, for example hash tables, are expected to satisfy fixed facts, called *invariants*. A state which satisfies appropriate invariants is a *consistent state*, and all other states are *inconsistent states*. Some operations involve the data temporarily being in an inconsistent state. A *race condition* occurs if another processor examines the data when in it is in an inconsistent state.

Mutual exclusion devices, henceforth *mutexes*, prevent race conditions by allowing at most one processor access at any time. Acquiring a mutex is a single, indivisible operation; all such operations are said to be *atomic*. If more than one processor tries to acquire a mutex simultaneously exactly one of them succeeds. Sections of code that require exclusive access to information are called *critical sections* and kept as short as possible.

Deadlock is possible if two or more processes require an overlapping set of mutexes. A minimal example is just two processes (A and B) both needing two mutexes α and β. If A holds α and is waiting for β while B holds β and is waiting for α then both processes block, waiting for a mutex to become available. The program never progresses beyond this point.

Semaphores are low-level devices for controlling access to critical sections. Two operations are supported: UP increments a semaphore atomically and DOWN decrements a semaphore atomically, unless it is 0. If a semaphore is zero all DOWN calls are suspended until an UP occurs, when exactly one DOWN call will proceed.

Monitors[16, 14] are higher level mutual exclusion devices and provide protected access methods for accessing a shared data object. Race conditions are impossible because only one monitor method, for one process, is active at any time. Operations that cannot be completed may be suspended within the monitor and resumed later.

2.2.2 Message passing

An alternative model of parallel computing is *message passing*. This model avoids race conditions by viewing each process as having a completely separate a set of resources. Processes share information by sending messages to each other. Deadlock is still possible, for example if processes A and B are waiting for a message from each other.

Deadlock occurs in a message passing system when it reaches a situation in which every individual process can communicate, but those with which it wishes to communicate block the communication. General results, for example those in Roscoe and Dathi's 1986 paper[34], can be used to prove with minimal effort that a large set of communication patterns are deadlock free.

Message passing scales to much larger numbers of processors than globally flat shared memory because there are no resources that must be shared globally. The model is easily implemented on multiple networked workstations, which many organisations already have. Some message passing libraries, for example MPI[40], are supported on a wide variety of hardware, including massively parallel machines.

[1] a generic name for 80386, 80486, pentium, pentium II, etc microprocessors

2.3 The complexity of parallel programming

The conditions that lead to errors in parallel programs due to violation of mutual exclusion, deadlock and other problems caused by multiple threads of control can be very complex. Adding debugging, for example including verbose messages about the progress of a computation, can make it impossible to replicate the conditions required for a bug to manifest itself.

Structured programming and object-orientated programming address complexity problems in sequential systems. Debugging is simplified by simplifying flow control; for example loops can be clearly identified. Improved modularity allows the implementation of an object to be analysed separately from its use. The additional complexity of parallel systems is not considered.

Two fundamental techniques have been applied to simplify the construction of parallel systems: simplification and analysis. Simplification avoids the problems that occur in complex circumstances by using simple communication and coordination structures, for example data parallel programming, SPMD (single program multiple data), BSP[21] (bulk synchronous parallel) and process farming. Analysis provides techniques to detect possible deadlocks and similar problems. Many analysis techniques are sufficiently general to prove properties of whole classes of communication patterns.

2.3.1 Analysis techniques

This subsection describes various common techniques used to analyse sequential imperative programs and then describes approaches to extending them to handle parallel systems.

Sequential program analysis

A common method of analysing sequential programs proves properties of pieces of programs and then composes the results to prove statements about larger pieces of the program. This is often called *assertional reasoning* because it asserts facts about the state of a program at specific places. Analysis of loops usually proves by induction a statement that the conditions at the top of a loop still apply after an iteration. Termination is usually proved by demonstrating an integer function that strictly decreases each iteration with a lower bound. The facts proved are usually based on the reasons the program is believed to work.

A related form of reasoning is *Hoare logic* which represents sequential programs as a sequence of *Hoare triples*, $\{P\}\ c\ \{Q\}$ where P is a *precondition*, c is some code and Q is a *postcondition*. If P applies before c is executed the Q will applied afterwards.

A calculus allows Hoare triples to be combined, using rules like $\{P\}\ c_1\{Q\}$; $\{Q\}\ c_2\ \{R\}$ implies $\{P\}\ c_1;\ c_2\ \{R\}$. This rule states that if c_1 starts establishes Q given P and c_2 establishes R given Q then c_1 followed by c_2 establishes R given P, under the usual interpretation of the notation. The analysis would apply equally to *any* interpretation of the same symbols.

A similar calculus applies to *precondition*, *postcondition* pairs. Using $[P, Q]$, where P is a precondition and Q is a postconditions, one rule allows $[P, Q]$ to the transformed into $[P, R]$; $[R, Q]$. As with Hoare triples there is a conventional interpretation of the notation but the analysis applies however the symbols are interpreted.

Extensions to parallel programs

Shared objects in parallel programs may be changed by several processes, which invalidates the assumptions that data not modified by an operation stays the same implicit in the methods of

analysis described in the previous subsection. This removes simple rules for composing established facts about the effect of sections of a program, except in severely circumscribed circumstances. A simple example is that if one section of code establishes X and another establishes Y given X, then joining them in a parallel environment is not reliable if another thread might make X false before Y is established.

Considering the activities of all threads together allows assertional reasoning to be applied to parallel systems[2, 36]. This can be prohibitively complex, especially if the processes are not synchronised (the usual case). Assertional reasoning does have the advantage that it is completely general and can prove more than just mutual exclusion, but also deadlock freedom, liveness, etc[1, 27].

Owicki and Gries provide an extension Hoare logic to cover parallel programs[26], allowing parallel composition subject to non-interference properties. Each thread is broken into a series of atomic steps with conditions with stated conditions between them, and it must be verified that the actions of other threads will not interfere with these properties. Thus exponential explosion of what has to be proved is remains, although the Owicki-Gries notation might make it easier to automated this analysis.

The rely/guarantee approach[6]that are relied upon and guaranteed **during** the operation to pre and post conditions. This allows analysis of what level of mutual exclusion is required in which places. It also simpler than Owicki-Gries and avoids many of the problems—combinations pre, rely, guarantee and post conditions can be combined and this can be used to combine threads in groups.

The dining philosophers problem[15, section 2.5, page 75][2] shows that an entire parallel system must be analysed; both forks and philosophers are deadlock free and removing any component makes the remainder of the system deadlock free. Despite this the complete dining philosophers system is not deadlock free.

Message passing avoids shared data, and thus race conditions and mutual exclusion requirements, but can be subject to subtle synchronisation problems if the communication pattern is complex. Assertions can be used to reason about these systems but process calculi are likely to yield the same results more simply.

2.3.2 Process calculi

There are several different process calculi, for example LOTOS[39], CCS[23], CSP[15, 33] and the π-calculus[24, 35]. All are defined in mathematical terms, use an idealised model and only analyse the pattern of communication. Most process calculi can theoretically be extended to analyse the value of variables too, albeit at the expense of reducing the tractability and ease of automating the analysis.

Very few systems actually implement the idealised models assumed by process calculi. Fortunately it is fairly easy to model more realistic models within the idealised model, and often possible to argue that the differences do not affect the validity of a result. All process calculi can be used to specify systems which are not deadlock free and can be used to analyse the possible causes of deadlock in systems which are not deadlock free.

All process calculi allow one to compose components in a relatively simple manner. The definition of a process almost always includes sufficient information to deduce that the dining

[2]This version of the dining philosophers problem does not include the butler, who prevents deadlock.

philosophers[15, 12, page 75] problem with one of the components removed is deadlock free, but the entire system can deadlock. Further they can show that the butler[3] prevents deadlock.

Process calculi allow one to analyse all the possible behaviours of a model for problems, including deadlock and live lock (also known as divergence). Extensions to the process calculi for handling time[7, 29] allow one to verify that programs meet deadlines. Process calculi have been used to prove bus negotiation protocols and that a fault tolerant railway signalling system meets safety criteria[4].

The unambiguous meaning, powerful analysis and proof techniques that process calculi provide make them useful for the specification, design and analysis of parallel programs. A. W. Roscoe and Dathi[34] use CSP to prove that a large class of communication patterns are deadlock free and a more systematic presentation with some further results can be found in *The theory and practice of concurrency* by A .W. Roscoe[33]. Automatic model checkers based on process calculi, such as the concurrency workbench[5], have solved many of the tractability problems of process calculi.

2.4 Simplified programming schemes

A large application, for example large scientific codes, can be millions of lines long and have very complex data dependencies. A number of approaches simplify parallel programming by restricting the design. Parallelising compilers, analogous to vectorising compilers, often automatically detect and exploit opportunities for data parallelism in existing programs.

2.4.1 Non-imperative programming languages

A number of programming languages, some of them for parallel systems, are *not* imperative programming languages. Parallel implementations of these languages avoid most of the well known problems by not supporting the problematic constructs.

A functional program is composed of function definitions for example `square x=x * x`. Recursion is used instead of loops, and only the values required are computed. There are no variables or control flow, so race conditions are impossible and mathematical analysis is simplified because there is no persistent state.

A UNITY program[3] consists of a state and set of guarded assignment statements which manipulate the state. Other systems that implement something similar include spreadsheets (when you change an input those cells that depend on that input are updated automatically).

At each step in a UNITY program *one* of the assignment statements is selected, and if its guard is true the state changes as specified by the assignment statement. A fairness condition states that in any infinite execution each assignment is chosen infinitely often. Since at most one assignment statement is selected at each step it could be argued that unity does *not* support parallel programming. The state stops evolving when a fixed point is reached when all the assignment statements do not change the state. Among other things UNITY supports the statement of invariants the state should satisfy after each step.

Neither functional programming languages nor UNITY are obviously readily implementable on conventional hardware without an interpreter or suitable frameworks for analysing imperative parallel programs. Even if the problem of adding control flow to programs without it is solved, which is still a research problem for functional programming languages, UNITY's global state makes efficient parallel implementation difficult.

[3]the butler only allows 4 of the 5 philosophers to reach the table at any time, preventing deadlock.

As a result neither UNITY nor functional programming languages are widely used for problems where performance is critical, for example computation fluid dynamics and the Hartree-Fock problem (which is discussed in chapter 9), which require a level of performance that is only available from imperative programming languages on conventional hardware.

2.4.2 Data parallelism

Data parallel programming languages, for example HPF[13, 37], model a single thread of control with statements applying to multiple data items (which are often implemented by distributing them across multiple processors). Methods of reasoning about sequential programs, for example claims about the values of variables at specific points, can be applied directly. Related methods of debugging, for example printing intermediate results to check their values, also work.

Automatically parallelising compilers, for example SUIF[9, 18], detect and exploit opportunities for data parallelism, often using techniques originally designed for vector machines. Hand parallelisation often produces better results because people can perform more extensive transformations than computer programs.

Vector machines use highly pipelined arithmetic—when the first operand has reached the second stage the next pair of operands is passed to the first stage. If there are n stages and the vectors are long then this is almost n times as fast using much less than n times the hardware. The low volume of vector hardware makes vector supercomputers very expensive. Some recent higher volume hardware, including the pentium 4, have pipelined floating point arithmetic.

Similar problems are tackled on clusters in a data parallel program by dividing the data between the n processors, as evenly as possible. If the expression is complex then the cost may be a complex function of the processor type and speed. Thus it is difficult to make effective use of multiple processors with a range types and speeds. One approach to this problem is to use dynamic load balancing algorithms which redistribute the load in the light of measured performance. The major disadvantage of these algorithms is that they use resources for a purpose which does not directly contribute to the desired result.

2.4.3 Bulk synchronous parallel

A bulk synchronous parallel program, henceforth BSP, proceeds in a series of supersteps each terminated by a global synchronisation. Communication takes place only at the end of a superstep. Deadlock is completely eliminated and the result is fairly efficient for a broad variety of applications.

BSP solves potential complexity problems by having no shared data within a superstep and making it easy to know what code each CPU is executing within each superstep. This allows one to analyse each superstep on a per processor basis, combine them for a complete analysis of a single superstep and compose supersteps simply[12].

2.4.4 Restricted communication patterns

Per Brinch Hansen et al have proposed the use of frameworks[10, 11] for various styles of application, for example n-body problems. The communication pattern of a template is fixed, and verified to be correct, for example using one of the formal methods discussed above. A new problem with the same communication requirements as the n-body problem can be implemented by just changing the details of the calculation.

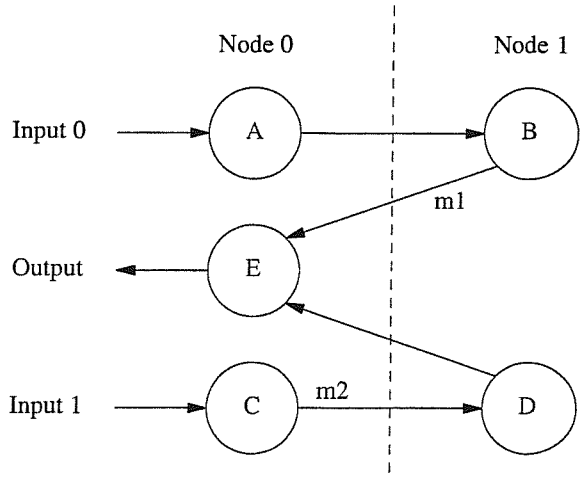


Figure 2.1: Deadlock scenario for an acyclic set of components due to aggregation on two nodes.

New programming languages, or libraries, which restrict the communication patterns to a class proved to have desirable properties are based on similar reasoning. Analysis of a fixed communication graph, a set of nodes and edges (pairs of nodes), does not suffice for these systems—instead a whole class of communication patterns has to be analysed.

Some languages allow the specification of a system which can deadlock and use static analysis to detect deadlock at compile time. The determination that a program is deadlock free is almost always definitely correct, albeit at the cost of not detecting the deadlock freedom of some deadlock free programs and computationally expensive analysis.

Common techniques for proving classes of systems are deadlock free include constructing a parameterised system that describes the behaviour of at most a few components, and then showing that adding a component to it results in another instance of the same system, usually with different parameters. It then suffices to show that this system has the claimed properties (for example, deadlock freedom).

2.4.5 Task parallelism

Algorithmic parallelism, sometimes called task parallelism, passes data among a network of processes which perform specific, and sometimes unique, transformations and integration of the incoming data. There is no shared data between components, although components can have private state. The components of a task parallel program can be analysed separately and simply composed to deduce an overall behaviour of the program, provided it is deadlock free.

The communication structure can be represented as a graph, with the nodes representing processes and an arc from node A to node B if node A sends a message to node B. It is frequently assumed there is no buffering between any pair of nodes–a destination node must be listening before a message can be sent. If a node is not listening then the sending node blocks until the message can be sent.

In this case the communication graph must be carefully designed to avoid a cycle of dependencies, which cause deadlock (all the nodes in the cycle are attempting to send a message, and all fail as result). Unless the deadlock freedom can be proved by reference to general results, for example one of the results in [34], a specific proof is required. This requires global analysis of the network, including possibly complex analysis to show that any cycles in the network are deadlock

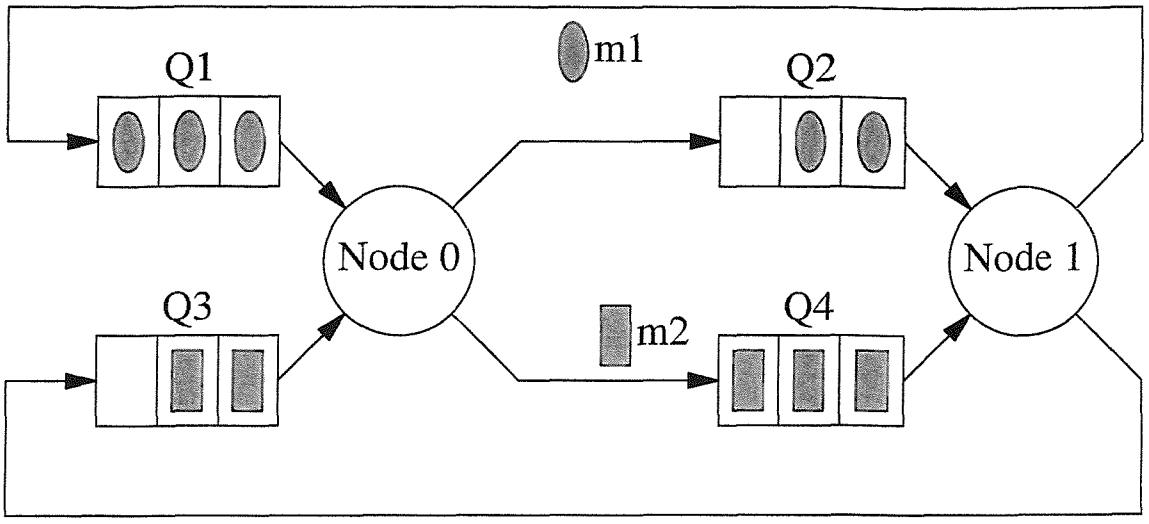


Figure 2.2: Deadlock scenario for two deadlock free cycles, adapted from [20].

free (the task parallel Hartree-Fock program, presented in chapter 9, necessarily contains several cycles).

Even if the network is deadlock free, aggregating multiple components into a single thread of control can introduce deadlock. The circles represent components and arrows the direction of a communication. A label on an arc represents an undeliverable message.

The communication structure of components A, B, C, D and E in figure 2.1 is acyclic and thus deadlock free[34]. Unfortunately this no longer applies when it is distributed on nodes 0 and 1 as shown in figure 2.1. Node 1 is trying to send message m1, which blocks because node 0 is trying to send message m2, and *vice versa*. Hence the system is deadlocked. This assumes the implementation is a loop that reads a message, processes it with the appropriate component body and then sends the results.

Buffers allow messages to be sent even if they cannot be delivered immediately, for example because the destination node is attempting to send a message, provided the storage requirements do not exceed their capacity, which is usually finite. A buffer capable of storing m1 on node 0, or m2 on node 1, would prevent the deadlock in figure 2.1.

If the buffer capacities are not carefully chosen, deadlock due to aggregation of components is still possible. In figure 2.2 neither the top cycle, processing elliptical jobs, nor the bottom cycle processing rectangular jobs, can deadlock in isolation due to the provision of sufficient buffering. Despite this neither m1 nor m2 can be delivered, because their destination buffers are full. Thus the overall system is deadlocked.

This problem can be avoided by careful choice of the buffer capacities, for example using the linear programming method in Liebeherr and Akyildiz's 1995 paper[20], which is intractable for some networks. Liebeherr and Akyildiz also present an efficient solution that works for a worst case subclass for their general solution. Unfortunately in many environments implementing carefully chosen buffer sizes is difficult, especially if the message sizes are not known in advance.

2.4.6 The design presented in this document

This document presents two variations on components with a slightly different design from input, process, output components. The input and output stages of an algorithmic parallel program

component are merged into a single parallel input and output phase. This avoids problems with cycles–all the outputs succeed because an output cannot prevent an input from occurring. The more subtle problems that arise due to the aggregation of components are also eliminated because they also depend on outputs preventing inputs from happening, which is not possible with a parallel input and output phase.

The model does not require centralised timing or synchronisation between more than two processes (as provided by synchronous communication channels). The language specified in [30] eliminates the need to add explicit communication to programs. The library described in chapter 6, and whose implementation is described in chapter 7, is a C library implementing the same design, allowing the use of established languages, libraries and sophisticated optimising compilers.

A program is composed of components that cycle continuously. Each component is structured in pseudo code as

$y = y_0$
forever
    Output y and read new x in parallel.
    Compute $y = f(x)$

Separate components can be analysed using the same techniques that are used for sequential programs. The tractability, simplicity and generality of these techniques, in the absence of the complexities of parallelism, make it fairly easy to prove the correctness of individual components and chains of components.

Two different kinds of programs are permitted: synchronous programs and asynchronous programs. Components in both types of program must listen to all their inputs after completing the calculation phase. At most one value may be read from any channel in any cycle. Given any input the computation phase must terminate within a finite time.

A synchronous program is multiple synchronous components in parallel. Synchronous components must read from all their inputs and write to all their outputs in any cycle. This allows one to argue that they satisfy certain synchronisation properties.

Asynchronous programs are multiple asynchronous components in parallel. Asynchronous components are only required to output to a (possibly empty) subset of their outputs and read from at least one input on any cycle. An asynchronous component must be prepared to read one input from any number of input channels before re-entering the computation phase. The subset of outputs used can vary from one cycle to another.

Both synchronous and asynchronous programs with any communication topology are deadlock free, without the need for any fairness condition. Deadlock is impossible because no component can refuse input or output unless it is busy calculating. The non-existence of pathological examples, where deadlock might occur only when very specific and unlikely timing applies, is *proved* in chapters 4 and 5 using CSP, which is a process calculus. The lack of restrictions on the behaviour components, and the unrestricted communication graph, suggests that there is no simple description of the behaviour of a set of components that is preserved when a component is added to it.

The deadlock freedom of a large class of programs over a general communication topology is valuable for embedded systems and other cases where correctness is of paramount importance. The

deadlock freedom of asynchronous programs is particularly significant because they are similar to programs composed of traditional input, process, output components. These are subject to many causes of deadlock that are difficult to eliminate in complex cases.

Chapter 3

CSP

Process calculi are usually used to analyse only the communication pattern of a parallel program. While process calculi can also be used, at least theoretically, to analyse the value of variables, this is usually intractable and unnecessary.

Process calculi are frequently used to analyse an entire class of communication patterns rather than a single program. If this is done the next step is usually to argue that a program, or programs implementable using a construction technique, have some property. Deadlock freedom is a common property to prove in this manner.

CSP (communicating sequential processes) is a process calculus that describes a concurrent system as a collection of interacting sequential processes. In CSP the term process covers both a single sequential process and the parallel composition of multiple sequential processes. CSP defines processes solely in terms of their observable behaviour, and this simplifies the proof. The term CSP in this document refers specifically to the process calculus defined in [15] and adopts the same notation.

3.1 Observable properties of processes

Each process has a finite alphabet, which is the set of events that it can influence. The alphabet of the process P is written αP. The behavior of a process is defined in terms of events, which occur singly. The trace of a process P is the sequence of events that P has engaged in. Traces can be written as a list of events in angle brackets, for example $\langle e_1, e_2, \ldots, e_n \rangle$ where $e_1, \ldots, e_n$ are events.

If the alphabet of a process is not obvious from context then the alphabet is written as a subscript, for example $\mathrm{RUN}_{\alpha P}$ is RUN with the same alphabet as P. The alphabet of a process P may also be specified explicitly by a statement, for example $\alpha P = \{a, b, c\}$ states that the alphabet of P is $\{a, b, c\}$.

If $\mathcal{T}$ is a trace after which P can engage in an infinite number of events all within a set of events $S \subseteq \alpha P$ then $\mathcal{T}$ is a divergence of $P\S$, the process P with all the events in S rendered invisible.

The failures and divergences model of CSP allows one to observe a process refusing to engage in sets of events and diverging. The following three sets completely describe a process[15, Definition D0].

1. The alphabet of P, αP, which is the set of events which P can influence.

2. Pairs of traces $\mathcal{T}$ and sets of events $\mathcal{E}$ that P might refuse to engage in after engaging in $\mathcal{T}$. Pairs $(\mathcal{T}, \mathcal{E})$ are *failures*. The failures of a process P is written as failures(P).

3. traces $\mathcal{T}$ after which P can diverge. The divergences of a process P is written as divs(P) below.

Sets of events $\mathcal{E}$ such that P might refuse to engage in as a first event are *refusals* of P. Showing that P and P' have the same alphabet, failures and divergences is a complete proof that $P = P'$. It is usually more convenient to use higher level proof rules, for example for $P \parallel P = P$ for any process P, to prove that pairs of processes are equal.

3.2 Basic Processes

CSP defines complex processes in terms of simpler processes. A sufficient set of basic process for constructing divergence-free processes is STOP and RUN. Proving that processes implemented in terms of themselves are well defined is not covered here, for details see section 3.9 of [15].

3.2.1 STOP

STOP refuses to engage in all events in its alphabet and has no divergences. If P can evolve into STOP them it is not deadlock free.

3.2.2 RUN

RUN never diverges and will engage in any event after any trace. The deadlock freedom proofs model the external inputs and outputs of a program as RUN processes which, by results below, have no effect and are therefore eliminated from the analysis.

3.3 Combining Processes

3.3.1 After

$P/\mathcal{T}$ for some trace $\mathcal{T}$ and process P behaves like P after P has engaged in the trace $\mathcal{T}$. If $\mathcal{T}$ is not a trace of P then $P/\mathcal{T}$ is not defined.

3.3.2 Prefixing

The process $e \rightarrow P$ engages in the event e and then behaves like P. A process which is defined recursively without any prefix can refuse any set of events after any trace and diverge after any trace (this is the global minimum of an appropriate interpretation of $<$).

3.3.3 Internal choice

The processes $P \sqcap Q$ behaves like either P or Q non-deterministically. The environment has no control over whether the process behaves like P or Q. This can be used to model unpredictable processes, for an output of an asynchronous component.

3.3.4 External Choice

The process $P \,\Box\, Q$ is willing to engage in a first event e_P of P and then behave like P/e_P or a first event e_Q of Q and then behave like Q/e_Q . The environment of $P \,\Box\, Q$is allowed to control this choice. Given any event e in the possible first events of P and Q then $(P \,\Box\, Q)/\langle e\rangle = (P/\langle e\rangle) \sqcap (Q/\langle e\rangle)$, i.e. a non-deterministic choice between P and Q occurs.

3.3.5 Synchronised Concurrency

The process $P \parallel Q$ engages in the events in αP if allowed to do so by P and the events in αQ if Q allows it. Given any event e in $\alpha P \cap \alpha Q$ then $P \parallel Q$ can engage in e if and only if both P and Q allow e to occur.

3.3.6 Re-labelling

Any process can be re-labeled by applying an invertible function f that renames the entire alphabet of P (to the same name or a different name). $f(P)$ behaves exactly like P except that all the events are transformed by f.

3.3.7 Filtering a trace

If $\mathcal{T}$ is a trace and $\mathcal{E}$ is a set of events then $\mathcal{T} \upharpoonright \mathcal{E}$ is the trace $\mathcal{T}$ with all events not in $\mathcal{E}$ removed. This operation is useful for making statements about the occurrence of a restricted set of events.

3.3.8 Joining two traces

If $\mathcal{T}_1$ and $\mathcal{T} - 22$ are traces then $\mathcal{T}_1 \frown \mathcal{T}_2$ is $\mathcal{T}_1$ followed by $\mathcal{T}_2$. This operation is used in statements like, if P is a deterministic process, $\mathcal{T}$ is a trace, e an event and $\mathcal{T} \frown \langle e\rangle$ is a trace of P then $P/\mathcal{T}$ cannot refuse e.

3.3.9 The length of a trace

If $\mathcal{T}$ is a trace then $\#\mathcal{T}$ is the number of events in $\mathcal{T}$. This operation is often used in combination with filtering to write the number of times an event has occurred (if e is an event and $\mathcal{T}$ a trace then $\#(\mathcal{T} \upharpoonright \{e\})$ is the number of times e occurs in $\mathcal{T}$).

3.3.10 Other operators

CSP also provides operators to send and receive messages, hide events, run P and Q in parallel without synchronisation (an event which can be accepted by P or Q is performed by either P or Q and not both). These operators are not used in this document.

3.3.11 The last occurrence in a trace

For any trace $\mathcal{T} = \langle e_1, e_2, \ldots\rangle$ and event e define

$$l(\mathcal{T}, e) = \begin{cases} 0 & \text{if } e \notin \mathcal{T} \\ \max\{i \mid e_i = e\} & \text{otherwise} \end{cases}$$

The indicies of the trace elements begin at 1 so $l(\langle e \rangle, e) = 1$. The main results in this thesis show that $l(\mathcal{T}, e) < l(\mathcal{T}, e)$ for a specific event e if a collection of synchronous or asynchronous components are deadlocked after a trace $\mathcal{T}$.

3.3.12 Parameterized templates

The analysis uses processes templates with parameters; substitution of specific parameters is required to convert these into specific processes as defined in [12]. Identically named parameters are assumed to remain the same within a proof; thus these parameterized templates are shorthand for specific processes. (These templates are not standard in CSP).

Example 1

myrun(S) = $\Box_{e \in S}\, e \rightarrow$ myrun(S) is equivilent to RUN$_S$ for any set of events S . Simply renaming events does not allow the choice of an arbitary number of events and thus there is no equivalent formulation in terms of renaming events.

If $S = \{a, b\}$ then myrun(S), as defined above and renamed to M is $M = (a \rightarrow M) \,\Box\, (b \rightarrow M)$.

3.4 Refinement and Deadlock freedom

If a process P *refines* a process Q if the behaviour of P is a subset of the behaviour of Q. Formally define $P = (f_P, d_P, a_P)$ where f_P = failures(P), d_P = divs(P) and a_P is the alphabet of P. P is a refinement of $Q = (f_Q, d_Q, a_Q)$ if and only if $a_P = a_Q$, $f_P \subseteq f_Q$ and $d_P \subseteq d_Q$.

If P is a refinement of Q then it is possible that $P/\mathcal{T}$ can refuse an event e if and only if $Q/\mathcal{T}$ can do so too. In particular if P can not refuse an event e then $(\langle\rangle, \alpha P) \notin$ failures(P) and thus P is **not** refined by STOP.

A process P is deadlock free[34] if and only if $\nexists \mathcal{T}$ such that $P/\mathcal{T}$ is refined by STOP. An equivalent formulation states that

P is deadlock free if and only if $\nexists \mathcal{T}$ such that $(\mathcal{T}, \alpha P) \in$ failures(P).

3.5 Additional properties of RUN

This section proves two results about RUN processes that are used in the proof itself. These results are simple extensions of the standard results about RUN processes and the proof is largely mechanical.

Result 1 (Combination of RUN processes)
RUN$_a$ || RUN$_b$ = RUN$_{a \cup b}$

Proof.

Using the rules in [15] calculate the alphabet, failures and divergences of RUN$_a$ || RUN$_b$ and RUN$_{a \cup b}$. Observe that both processes have the same sets of failures, divergences and alphabet, and thus are identical.

$$
\begin{aligned}
\alpha(\mathrm{RUN}_a \parallel \mathrm{RUN}_b) &= (a \cup b) \\
&= \alpha\mathrm{RUN}_{a\cup b} \\
\mathrm{divs}(\mathrm{RUN}_a \parallel \mathrm{RUN}_b) &= \Big\{ s \frown t \mid t \in (a \cup b)^* \\
&\quad \wedge \Big(((s \restriction a) \in \emptyset \wedge (s \restriction b) \in b^*) \\
&\quad \vee ((s \restriction b) \in \emptyset \wedge (s \restriction a) \in a^*) \Big) \Big\} \\
&= \{ s \frown t \mid t \in (a \cup b)^* \\
&\quad \wedge ((\mathrm{false} \wedge \mathrm{true}) \vee (\mathrm{false} \wedge \mathrm{true})) \} \\
&= \Big\{ s \frown t \mid \mathrm{false} \Big\} = \emptyset \\
&= \mathrm{divs}(\mathrm{RUN}_{a\cup b}) \\
\mathrm{fail}(\mathrm{RUN}_a \parallel \mathrm{RUN}_b) &= \{ s, (X \cup Y) \mid s \in (a \cup b)^* \\
&\quad \wedge (s \restriction a, X) \in \{ s, \emptyset \mid s \in a^* \} \\
&\quad \wedge (s \restriction b, Y) \in \{ s, \emptyset \mid s \in b^* \} \} \\
&= \{ s, (X \cup Y) \mid s \in (a \cup b)^* \wedge (s \restriction a) \in a^* \wedge X = \emptyset \\
&\quad \wedge (s \restriction b) \in b^* \wedge Y = \emptyset \} \\
&= \{ s, \emptyset \mid s \in (a \cup b)^* \wedge \mathrm{true} \wedge \mathrm{true} \} \\
&= \{ s, \emptyset \mid s \in (a \cup b)^* \} \\
&= \mathrm{fail}(\mathrm{RUN}_{a\cup b})
\end{aligned}
$$

Thus $\mathrm{RUN}_a \parallel \mathrm{RUN}_b$ and $\mathrm{RUN}_{a\cup b}$ have the same alphabet, failures and divergences. These sets completely specify a process[15, Definition D0]. Thus $\mathrm{RUN}_a \parallel \mathrm{RUN}_b = \mathrm{RUN}_{a\cup b}$. □

The next result shows that any process P is unaffected by synchronation with a RUN process whose alphabet is any subset of the alphabet of P. This result combined with the previous result makes it easy to eliminate the RUN processes.

Result 2 (Generalisation of $P \parallel \mathrm{RUN}_{\alpha P} = P$)
For all processes P and $S \subseteq \alpha P$ $P \parallel RUN_S = P$

Proof.

Using the rules in [15] calculate the alphabet, failures and divergences of $P \parallel \mathrm{RUN}_S$ and P. Observe that both processes have the same sets of failures, divergences and alphabet, and thus are identical.

$$
\begin{aligned}
\alpha(P \parallel \mathrm{RUN}_S) &= (\alpha P \cup S) \\
&= \alpha P \qquad (S \subseteq \alpha P \text{ so } \alpha P \cup S = \alpha P) \\
\mathrm{divs}(P \parallel \mathrm{RUN}_S) &= \{ t \in \mathrm{divs}(P) \mid (t \restriction S) \in \mathrm{traces}(\mathrm{RUN}_S) \} \\
&\quad \cup \{ t \in \mathrm{divs}(\mathrm{RUN}_5 S) \mid (t \restriction \alpha P) \in \mathrm{traces}(P) \} \\
&= \{ t \in \mathrm{divs}(P) \mid (t \restriction S) \in S^* \} \\
&\quad \cup \{ t \in \emptyset \mid t \in \mathrm{traces}(P) \}
\end{aligned}
$$

$$
\begin{aligned}
&= \{t \in \text{divs}(P) \mid \text{true}\} \cup \emptyset \\
&= \text{divs}(P) \\
\text{fail}(P \parallel \text{RUN}_S) &= \{(s, (X \cup Y)) \mid s \in (\alpha P \cup S)^* \\
&\quad \wedge (s \upharpoonright \alpha P, X) \in \text{fail}(P) \\
&\quad \wedge (s \upharpoonright S, Y) \in \{(v, \emptyset) \mid v \in S^*\}\} \\
&= \{s, (X \cup Y) \mid s \in (\alpha P \cup S)^* \wedge (s \upharpoonright \alpha P, X) \in \text{fail}(P) \\
&\quad \wedge Y = \emptyset\} \\
&= \{(s, X) \mid s \in \alpha P^* \wedge (s, X) \in \text{fail}(P)\} \\
&\qquad\qquad (X \cup \emptyset = X \text{ and } S \subseteq \alpha P \text{ so } \alpha P \cup S = \alpha P) \\
&= \text{fail}(P)
\end{aligned}
$$

Thus if $S \subseteq \alpha P$ then $P \parallel \text{RUN}_S = P$. □

3.6 Environment

The environment of a program consists of its input channels (keyboards, in-bound network connections, etc) and its output channels (displays, out-bound network connections, etc). An input channel is always prepared to allow reading and an output channel is always prepared to engage in an output.

Since events make no distinction between input and output for any event e the process representing an external input, $\mathsf{in}(e)$, and an external output, $\mathsf{out}(e)$, can be defined as

$$\mathsf{in}(e) = \mathsf{out}(e) = \mathsf{inout}(e) = e \rightarrow \mathsf{inout}(e) = \text{RUN}_{\{e\}}$$

Since timing and the order of events is not used by the analysis the possible need to wait is irrelevant—the ability to perform another event is sufficient.

3.7 Summary

The elements of CSP used in the next two chapters have been presented. A process was defined and used to prove a couple of extensions to standard results about RUN.

Chapter 4

Synchronous programs

4.1 Modeling a single synchronous process

A synchronous program is a number of synchronous processes operating in parallel. The component processes must operate as described in description 1

Description 1

A synchronous process is a process that

- Reads a value from *all* its input channels before starting a cycle
- Writes a value to *all* its output channels before starting a cycle
- Finishes its computation step within a finite time

The process code may then be written as

$x = x_0$
while (global termination condition not met)
    compute $y = f(x)$
    output y using *all* output channels
      and read next x using *all* input channels in parallel

or any equivalent construction. The details of the calculation and I/O depends on the environment, problem and programming language.

The synchronization requirements are surprisingly loose as shown by result 4 below. Keyboards, displays and so forth are treated as external sources of infinitely readable inputs, or infinitely writable outputs.

Input and output on any channel must be synchronized and this is where the possibility of deadlock arises. Proof that deadlock is impossible requires a precise description of a synchronous program and thus a synchronous component. A synchronous component is defined as the $S(I, O)$ process in 4.1, where I is the set of input channels, and O is the set of output channels.

$$\begin{aligned} o(c) &= c \to s \to o(c) & \alpha o(c) &= \{c, s\} \\ i(c) &= c \to s \to i(c) = o(c) & \alpha i(c) &= \{c, s\} \\ S(I, O) &= \mathop{\Big\|}_{c \in I \cup O} o(c) & \alpha S(I, O) &= I \cup O \cup \{s\} \end{aligned} \tag{4.1}$$

Here $i(c)$ inputs on channel c and $o(c)$ outputs on channel c. All the events in I correspond to reading an input and all the events in O to writing an output. Since input and output are both compulsory and represented as an event input and output processes are identical.

$S(I, O)$ is a cyclic process. In each cycle $S(I, O)$ first reads an input from all the inputs in I and writes an output to all outputs in O. Then it engages in s, it performs a calculation (not modelled) and then the cycle starts again. After the compute stage all the $o(c)$ processes return to their initial state and thus a new cycle begins.

Lemma 1 (Acceptance of a communication)
$S(I, O)/\mathcal{T}$ accepts e if and only if $l(\mathcal{T}, e) \leq l(\mathcal{T}, s)$ for all traces $\mathcal{T}$ of $S(I, O)$ and events $e \in I \cup O$.

Proof.

$o(e) = e \rightarrow s \rightarrow o(e)$ is the only component of $S(I, O)$ with e in its alphabet. Thus it is sufficient to consider $o(e)$ after the trace $\mathcal{T}' = \mathcal{T} \upharpoonright \{e, s\}$, the trace $\mathcal{T}$ restricted to $\alpha o(e)$. Clearly $S(I, O)/\mathcal{T}$ accepts e if and only if $o(e)/\mathcal{T}'$ accepts e and $l(\mathcal{T}, e) \leq l(\mathcal{T}, s)$ if and only if $l(\mathcal{T}', e) \leq l(\mathcal{T}', s)$.

If $\mathcal{T}' = \langle\rangle$ then $l(\mathcal{T}', e) = l(\mathcal{T}', s) = 0$ and $o(e)/\mathcal{T}' = o(e)$ which accepts e (and refuses s) as required by the result.

Otherwise since $\mathcal{T}'$ is composed only of s and e events if and only if $l(\mathcal{T}', e) < l(\mathcal{T}', s)$ the last event of $\mathcal{T}'$ is s and thus $o(e)/\mathcal{T}'$ accepts e (and refuses s). If $l(\mathcal{T}', e) > l(\mathcal{T}', s)$ then the last event of $\mathcal{T}'\mathsf{T}$ is e and thus $o(e)/\mathcal{T}'$ refuses e (and accepts s). Equality is clearly impossible if $\mathcal{T}' \neq \langle\rangle$. □

Corollary 2 (Refused implies used)
If $S(I, O)/\mathcal{T}$ refuses $e \in I \cup O$ then $e \in \mathcal{T}$

Proof.

If $e \notin \mathcal{T}$ then $l(\mathcal{T}, e) = 0$ and thus $l(\mathcal{T}, e) \leq l(\mathcal{T}, s)$ which implies that $S(I, O)/\mathcal{T}$ accepts e by lemma 1. □

4.2 A Synchronous Program is deadlock free

The simplest explanation of why a synchronous program must be deadlock free is that initially every process must engage in a synchronised send on its output and read from its input channels. This must succeed because the internal output channels are connected to other synchronous processes that must be trying to read from them. External channels are assumed to be readable or writable as appropriate; if this is not true then the system is obviously liable to deadlock. The same argument applies on subsequent cycles.

CSP[15] can be used to prove the absence of deadlock. If the synchronous processes are $S_1, \ldots, S_n$ they share s so special measures must be taken to prevent them from synchronising on s, which is meant to be internal to each process. The method adopted here is to apply a relabelling function $f_i(e)$ to the process so each components has a different s event. Define f_i as

$$f_i(e) = \begin{cases} s_i & \text{if } e = s \\ e & \text{otherwise} \end{cases}$$

and a synchronous program as $SP = \|_{i=1,\ldots,n} f_i(S_i)$. f_i refers to this specific function in the results in this section. Define I_i as the input channels and O_i as the output channels of the ith

process. Since f_i transforms the alphabet of S_i as well as the events $\alpha(f_i(S_i)) = I_i \cup O_i \cup \{s_i\}$ and thus $f_i(S_i)$ and $f_j(S_j)$ can only synchronise on channels that they share for any $i \neq j$.

Lemma 3 (Deadlock requires all components to be blocked by another)
If $SP/\mathcal{T}$ is refined by STOP, *where SP is as defined above and $\mathcal{T}$ is any trace of SP, then for all i there exists $j \neq i$ and an event $e \in \alpha S_i \cap \alpha S_j$ such that $S_i/\mathcal{T} \upharpoonright \alpha S_i$ accepts e and $S_j/\mathcal{T} \upharpoonright \alpha S_j$ refuses e.*

Proof.

Define an event e as shared if $\exists\, i, j$ such that $i \neq j$ and $e \in \alpha S_i$ and $e \in \alpha S_j$. Any event e which is not shared is unshared. Assume for a contradiction that $SP/\mathcal{T}$ is refined by STOP and $S_i' = S_i/\mathcal{T} \upharpoonright \alpha S_i$ and $f_i(o(e))/\mathcal{T} \upharpoonright \{e, s_i\} = s_i \rightarrow f_i(o(e))$ for all shared events e.

Let $e' \in \alpha S_i$ be any unshared event such that $f_i(o(e'))/\mathcal{T} \upharpoonright \{e', s_i\}$ refuses s_i . If no such e' exists then the component processes of S_i' accept s_i and thus so does $SP/\mathcal{T}$. Otherwise the only component of $SP/\mathcal{T}$ with e' in its alphabet is $f_i(o(e))/\mathcal{T} \upharpoonright \{e', s_i\}$ which accepts e' and thus SP accepts e'. □

Lemma 4 (Deadlock requires a cycle)
If $SP/\mathcal{T}$ is refined by STOP *then $\exists\, n_0, n_1, \ldots, n_{k-1}$ such that there exists $e_i \in \alpha S_{n_i} \cap \alpha S_{n_{i\oplus 1}}$ such that $S'_{n_{i\oplus 1}}$ accepts e and $S'_{n_{i\oplus 1}}$ refuses e for all i where $S_j' = S_j/\mathcal{T} \upharpoonright \alpha S_j$ and $\oplus$ is addition modulo k.*

Proof.

Define a path of length k in a directed graph $G = (V, E)$ as a sequence of nodes $v_0, v_1, \ldots, v_{k-1}$ (not all necessarily distinct), such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < k$.

Let $G = (V, E)$ be the directed graph where node i represents the S_i component of SP, for all $0 \leq i < n$ and $(i, j) \in E$, an edge from node i to node j, if and only if there exists an event $e \in \alpha S_i \cap \alpha S_j$ such that S_i' accepts e and S_j' refuses e, where $S_k' = S_k/\mathcal{T} \upharpoonright \alpha S_k$. A cyclic path in G corresponds to a cycle of blocked processes as required by the result.

Claim by induction that for all $k > 0$ there is a path length k in G. Paths of length 1 consist of a single node and so clearly exist.

For all i there $\exists\, j \neq i$ such that $(i, j) \in E$ by lemma 3. Thus if $v_0, \ldots, v_{k-1}$ is a length k path, which exists by the induction hypothesis, there $\exists\, v' \in V$ such that $v \neq v_{k-1}$ and $v_0, \ldots, v_{k-1}, v'$ is a length $k+1$ path in G.

Thus by induction there exists a length $n+1$ path $v_0, \ldots, v_n$. Since there are only n nodes $\exists\, i < n$ such that $v_n = v_i$. $v_i, \ldots, v_{n-1}, v_i$ is a cyclic path in G. □

The existence of a cycle of components that each prevent the progress of the next component in the cycle means it is sufficient to show that such a cycle can not exist to prove deadlock freedom. The main theorem of this section shows that no such cycle can exist and thus SP is deadlock free.

Theorem 3 (A synchronous program is deadlock free)
$SP = \|_{i=1,\ldots,n} f_i(S_i)$ where $S_1, \ldots, S_n$ are independent synchronous processes, as defined above, is deadlock free.

Proof.

Assume for a contradiction that $\mathcal{T}$ is a trace of SP such that $SP/\mathcal{T}$ is refined by STOP.

Define $S_i' = f_i(S_i)/(\mathcal{T} \upharpoonright \alpha f_i(S_i))$, i.e. component S_i after SP has engaged in $\mathcal{T}$. In a deadlocked situation all S_i' must refuse s_i for all i and therefore must be waiting for either at least one input or output.

Lemma 4 shows that there is at least one cycle of components $n_0, \ldots, n_{k-1}$ and corresponding events $e_i \in \alpha f_{n_i}(S_{n_i}) \cap \alpha f_{n_{i\oplus 1}}(S_{n_{i\oplus 1}})$ such that $S'_{n_{i\oplus 1}}$ prevents S'_{n_i} from engaging e_i for all i, where $\oplus$ is addition modulo k.

Thus if $SP/\mathcal{T}$ is refined by STOP then $SP/\mathcal{T}$ contains a cycle of components such that each blocks the next one. Deriving a contradiction from the assumption that $\mathcal{T}$ does this clearly proves the result.

Assume without loss of generality, by renumbering components if required, that $n_0, \ldots, n_k = 0, 1, ..., k-1$ and thus $S_{n_i} = S_i$ and $S'_{n_i} = S'_i$ for all i. The fact that $SP/\mathcal{T}$ is refined by STOP implies that $e_i \in \mathcal{T}$ for all i (otherwise $S'_{i\oplus 1}$ could not refuse e_i).

Thus the acceptence of e_i by S'_i shows that $l(\mathcal{T}, e_i) < l(\mathcal{T}, s_i)$ and the refusal of e_i by $S'_{i\oplus 1}$ shows that $l(T, s_{i\oplus 1}) < l(\mathcal{T}, e_i)$. Together these imply that $l(\mathcal{T}, s_{i\oplus 1}) < l(\mathcal{T}, s_i)$ and the analysis obviously applies for all i. Hence, formally by induction, $l(\mathcal{T}, s_0) < l(\mathcal{T}, s_0)$ therefore no blocked cycle is possible. Since a trace $\mathcal{T}$ such that $SP/\mathcal{T}$ is refined by stop requires a blocked cycle no such trace can exist. □

4.3 A synchronous program is live

This section proves a *liveness* property of synchronous programs; here, liveness means that a synchronous program must output something eventually, provided at least one component has an external output. This is proved by showing a bound on the number of cycles computed by two components, whether they are connected directly or indirectly. If a synchronous component with an external output must have completed a cycle, then an output must have been generated.

It is easy to determine limits on the number of cycles performed by two directly connected processes; induction suffices to prove bounds for indirectly connected processes.

For the benefit of the results in this section define

cycles$(S_i, \mathcal{T}) = \#(\mathcal{T} \upharpoonright \{s_i\})$ where $\mathcal{T}$ is the trace of the synchronous program containing S_i. Since a synchronous component S_i engages in s_i at most once in any cycle cycles$(S_i, \mathcal{T})$ is the number of cycles S_i has completed after SP has engaged in $\mathcal{T}$.

Lemma 5 (Cycle count bounds for directly connected processes)
If S_1 and S_2 are two synchronous processes connected by a channel c and the synchronous program, SP, containing them has engaged in the trace $\mathcal{T}$ then cycles$(S_2, \mathcal{T}) - 1 \leq$ cycles$(S_1, \mathcal{T}) \leq$ *cycles*$(S_2, \mathcal{T}) + 1$.

Proof.

The only components of S_1 and S_2 with e in their alphabet are $O_1 = e \rightarrow s_1 \rightarrow O_1$ and $O_2 = e \rightarrow s_2 \rightarrow O_2$, and thus the problem can be reduced to the behaviour of $O_1 \parallel O_2$. The rules in [15] show that $O_1 \parallel O_2 = P$ where $P = e \rightarrow (s_1 \rightarrow s_2 \rightarrow P) \,\square\, (s_2 \rightarrow s_1 \rightarrow P)$.

Formally by induction, if $\mathcal{T}' \frown \langle e \rangle$, where $\mathcal{T}'$ is a trace of P and $e \in \alpha P$ is an event, then $\#(\mathcal{T} \upharpoonright \{s_1\}) = \#(\mathcal{T} \upharpoonright \{s_2\})$.

A direct consequence of this is that if $T' = T'' \frown \langle e, s_i \rangle$, where $\mathcal{T}''$ is a trace of P, then $\#(\mathcal{T}' \restriction \{s_1\}) = \#(\mathcal{T}' \restriction \{s_2\}) - 1$ and $P/\mathcal{T}'$ will refuse any event except $s+2$. Similarly if $\mathcal{T}' = \mathcal{T}'' \frown \langle e, s_2 \rangle$, where $\mathcal{T}''$ is a trace of P, is a trace of P then $\#(\mathcal{T}' \restriction \{s_1\}) = \#(\mathcal{T}' \restriction \{s_2\}) + 1$ and $P/\mathcal{T}'$ will refuse any event except s_1. Thus if $\mathcal{T}'$ is a trace of P then $\#(\mathcal{T}' \restriction \{s_2\}) - 1 \leq \#(\mathcal{T}' \restriction \{s_1\}) \leq \#(\mathcal{T}' \restriction \{s_2\}) + 1$.

If $\mathcal{T}$ is a trace SP then $\mathcal{T} \restriction \alpha(O_1 \parallel O_2)$ must be a trace of $O_1 \parallel O_2 = P$ because SP is a parallel composition of many processes including O_1 and O_2. Thus $\#(\mathcal{T}' \restriction \{s_2\}) - 1 \leq \#(\mathcal{T}' \restriction \{s_1\}) \leq \#(\mathcal{T}' \restriction \{s_2\}) + 1$ proves the result. □

Result 4 (Cycle count bounds for indirectly connected processes)
If the shortest route for S_i to S_j involves k communications and the synchronous program, SP, containing them has engaged in the trace $\mathcal{T}$ then

$$\text{cycles}(S_j, \mathcal{T}) - k \leq \text{cycles}(S_i, \mathcal{T}) \leq \text{cycles}(S_j, \mathcal{T}) + k$$

Proof.

This result is proved by induction on k.

Induction step
Assume by induction that the result holds for $k = n$. If S_j is $n+1$ communication events away from S_i then there exists S_m such that

- S_m is k communication events away from S_i
- S_m sends a result to S_j over channel c.

By the induction hypothesis $\text{cycles}(S_m, \mathcal{T}) - n \leq \text{cycles}(S_i, \mathcal{T}) \leq \text{cycles}(S_m, \mathcal{T}) + n$. Lemma 5 shows that $\text{cycles}(S_m, \mathcal{T}) - 1 \leq \text{cycles}(S_j, \mathcal{T}) \leq \text{cycles}(S_m, \mathcal{T}) + 1$. Thus

$$\begin{aligned} &\text{cycles}(S_m, \mathcal{T}) - n \leq \text{cycles}(S_i, \mathcal{T}) \leq \text{cycles}(S_m, \mathcal{T}) + n \\ \Rightarrow &\text{cycles}(S_j, \mathcal{T}) - n - 1 \leq \text{cycles}(S_i, \mathcal{T}) \leq \text{cycles}(S_j, \mathcal{T}) + 1 + n \end{aligned}$$

Base case
If $k = 1$ the result is lemma 5. Thus the result follows by mathematical induction. □

Result 4 proves that a synchronous program is live. If an external input I is connected to a component S_i, an external output O is connected to S_o, and there is a path involving n communications from S_i to S_o then result 4 requires at least one output before the $n+2$nd input from I is read. This guarantee of an output given sufficient input is a liveness property, because it guarantees that something good will happen.

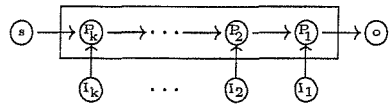
The bounds are tight, which means they can actually be achieved. A single example, which can be scaled to any value of n, that achieves the stated bound suffices to prove this result.

Result 5 (Tightness of upper bound)
The upper bound of result 4 is tight.

Proof.

This can be shown by manipulating the number of inputs read from the external input channels. The external output ⓞ is assumed to the always writable and the source ⓢ always readable.

For any $k \geq 1$ an example of a process that achieves the maximum stated in result 4 looks like



where $\textcircled{i_n}$ is an input channel which has been read n times. The synchronous program is the P_i processes in the box in the figure.

Claim 1 *Claim that P_j has cycled exactly j times provided sufficient input was readable from P_{j-1}*

Proof of Claim. On each cycle P_j must read a value from I_j and thus has cycled a maximum of j times. For $j \geq 2$ assume by the induction hypothesis that P_{j-1} cycles $j-1$ times. Since P_{j-1} completes $j-1$ cycles it must send $j-1$ values to P_j and be willing to send a jth values to P_j. This proves the induction step.

The base case, $j = 1$, is trivial because $\textcircled{o}$ always accepts input. The claim follows by mathematical induction.

Claim 2 *Claim that P_{j+1} will receive $j+1$ messages from P_j.*

Proof of Claim. For any $j < k$ assume by the induction hypothesis that P_j has received $j+1$ messages from P_{j+1}. Since this is enough input claim 1 states that P_j will cycle exactly j times. Thus P_{j-1} will receive j values from P_j (it can do this by claim 1). This proves the induction step.

The base case of the claim $j = k$ is trivial because the source $\textcircled{s}$ is always readable. The claim follows by mathematical induction.

Given any $j \leq 1$ claim 1 implies that it will cycle j times given enough input and claim 2 implies it will receive enough input.

Thus, in particular, P_j will cycle j times and P_1 will cycle once. The number of communication steps between P_j and P_1 is $j-1$ thus the upper bound on the number of cycles claimed by result 4 is $j-1+1 = j$ cycles. Since this is achieved the upper bound stated is tight. □

Corollary 6 (Tightness of lower bound)
The lower bound of result 4 is tight.

Proof.

If $\mathcal{T}$ is a trace of a synchronous program $S\mathsf{P}$ such that there exists components i and j separated by n communications such that $\text{cycles}(P_i, \mathcal{T}) = \text{cycles}(P_j, \mathcal{T}) + n$, which is possible by result 5, it also satisfies the lower bound because $\text{cycles}(P_j, \mathcal{T}) = \text{cycles}(P_i, \mathcal{T}) - n$ and the direction of the communications involved is irrelevant. □

The tightness of the upper and lower bounds shows how loose the synchronisation in a synchronous program is in reality–two components of a synchronous program may have completed significantly different numbers of cycles, provided they are sufficiently separated.

Once the limits are reached the synchronous program is unable to make any progress until the components that are preventing further progress perform another cycle. This effect might lead to the components being "clocked" by the component with the longest cycle time, which could be due to its need to read from a particular slow external input, for example a keyboard. Asynchronous programs, discussed in the next chapter, avoid this problem by insisting on only a single input before a cycle can begin.

4.4 Summary

A tighly synchronised, and very limiting, parallel program design was presented and shown to have deadlock freedom and liveness properties. The liveness properties can lead to a state where

the system is unable to proceed until a given event, which could be a serious problem in a system which must degrade gracefully.

Chapter 5

Asynchronous programs

An asynchronous program is a connected, finite collection of components. Each component outputs to some, possibly empty, subset of its output channels and reads input from at least one input channel before entering the computing phase. No fairness condition is required for the deadlock freedom result. A program must have a least one infinitely readable external input. This rules out programs that require a stimulus before anything happens and can never receive one, which are deadlocked.

A component is described in English as

Description 2

An asynchronous component is a component that

- Reads a value from one or more of its input channels before starting a cycle
- Can start a cycle based on one value from any non-empty subset of its inputs.
- Writes a value to all of a subset of its output channel before starting a cycle
- Finishes its computation step within a finite time

Each component is structured in pseudo code as

$y = y_0$
$o = o_0$
forever
    output y using channels in o
        and read x using at least one input channel in parallel
    compute $(y, o) = f(x)$

The details of the calculation are implementation dependent and not analysed here. A component must be willing to read all of its inputs in each cycle–a component is not allowed to choose some subset of its inputs and only react to their readability. If receiving inputs and sending outputs is not performed in parallel the deadlock freedom result does not apply.

These programs operate reliably because a component can only ignore a message for a finite time before reading it, unless another message is available. Since it only takes one message to start a new cycle the system will continue to cycle indefinitely. Many fairness conditions will ensure all messages are read within a finite time.

5.1 Environment

The environment of a program consists of its input channels (keyboards, in-bound network connections, etc) and its output channels (displays, out-bound network connections, etc). An input channel is always prepared to allow reading and an output channel is always prepared to engage in an output.

Since events make no distinction between input and output for any event c, representing an external communication, an external input, $\mathsf{in}(c)$, and external output, $\mathsf{out}(c)$, can be defined as

$$\mathsf{in}(c) = \mathsf{out}(c) = \mathsf{inout}(c) = c \rightarrow \mathsf{inout}(c) = RUN_{\{c\}}$$

5.2 Modelling a single component

An asynchronous component is defined as $A(I, O)$ in (5.1), where I is the set of inputs, O the set of outputs available. The process is cyclic and can use a different selection of outputs in each cycle.

$$\begin{aligned}
t(c) &= c \rightarrow s \rightarrow t(c) \sqcap s \rightarrow t(c) & \alpha t(c) &= \{c, s\} \\
i(c) &= c \rightarrow s \rightarrow i(c) \,\Box\, s \rightarrow i(c) & \alpha i(c) &= \{c, s\} \\
l(I) &= \Box_{c \in I}\, c \rightarrow l'(I) & \alpha l(I) &= I \cup \{s\} \\
l'(I) &= \Box_{c \in I}\, c \rightarrow l'(I) \,\Box\, s \rightarrow l(I) & \alpha l'(I) &= I \cup \{s\} \\
A(I, O) &= \|_{c \in I}\, i(c) \;\|_{c \in O}\, t(c) \;\|\; l(I) & \alpha A(I, O) &= I \cup O \cup \{s\}
\end{aligned} \tag{5.1}$$

$t(c)$ is an output via channel c and $i(c)$ possibly reads a value from channel c. Outputs choose whether or not to perform an output and inputs are required to accept an input if one is sent. $l(I)$ implements the *one or more* condition, as explained below, and $A(I, O)$ is the component. The computation phase only affects the timing, which is not incorporated into the model, and is assumed to occur between the s event and the start of the next cycle. Why does $A(I, O)$ satisfy the requirements of description 2?

If an output is not being used $t(c)$ engages in s, which marks the end of the communication phase of a cycle. If an output is in use $t(c)$ does the output and then engages in s. The use of a non-deterministic choice allows the compute phase to choose which behaviour will occur in the next cycle.

The *one or more* input condition is implemented here by using a lock (the $l(I)$ process). Until an input has been read $l(I)$ prevents all $A(I, O)$ from engaging in s. Thus $A(I, O)$ cannot complete the communication phase, by engaging in s, until an input has been read. Reading one input from channel c causes $l(I)$ to turn into $l'(I)$ which is willing to engage in any input event or s, allowing the $A(I, O)$ to enter its compute phase.

It is impossible to input or output more than one value on any channel per cycle because the input and output processes insist on engaging in s before performing another input or output. After engaging in s, and finishing the computation, the components of $A(I, O)$ are in their initial state and the cycle begins again (with possibly different internal choices about which outputs to use).

Lemma 7 ($A(I, O)$ **is cyclic**)
$A(I, O)/\mathcal{T} = A(I, O)$ if $\exists$ a trace $\mathcal{T}' \in (I \cup O)^*$ such that $\mathcal{T} = \mathcal{T}' \frown \langle s_i \rangle$.

Proof.

All the components of $A(I, O)$ return to their original state after engaging in s. □

5.3 An asynchronous program is deadlock free

A program is deadlock free because no message can be refused indefinitely by any component, unless another message is read. The conditions require external channels to complete input or output (depending on the direction of the channel) in a finite time. Thus every message will be delivered within a finite time. The proof derives a contradiction about any trace leading to a deadlocked state of an asynchronous program.

Since only one message is required for a component to begin a cycle this shows that the compute phase can be entered in response to a message within a finite time. Thus programs are deadlock free. The CSP analysis derives a contradiction from deadlock.

If the components are $A_1, \ldots, A_n$ they share s so special measures must be taken to prevent them from synchronising on this event, which is meant to be internal to each component. The method adopted here is to apply a relabelling function $r_i(e)$ to the components to differentiate each components' s event. Define r_i as

$$r_i(e) = \begin{cases} s_i & \text{if } e = s \\ e & \text{otherwise} \end{cases}$$

and a program as

$$\begin{aligned} AP &= \mathop{\Big\|}_{i=1,\ldots,n} r_i(A_i) \quad \Big\| \quad \mathop{\Big\|}_{c \in \text{external channels}} \text{inout}(c) \\ &= \mathop{\Big\|}_{i=1,\ldots,n} r_i(A_i) \parallel \text{RUN}_{\text{external channels}} \qquad \text{by result 1} \\ &= \mathop{\Big\|}_{i=1,\ldots,n} r_i(A_i) \qquad \text{by result 2} \end{aligned} \tag{5.2}$$

Theorem 6 (Deadlock freedom of AP)
There does not exist a trace $\mathcal{T}$ such that $AP/\mathcal{T}$ is refined by STOP.

Proof.

For all traces $\mathcal{T}$ and i, component i in $AP/\mathcal{T}$ must be in one these of 3 states

1. able to engage to s_i.
2. waiting for an input from any of its inputs.
3. waiting for an output to complete (and possibly an input too).

Assume for a contradiction that $\mathcal{T}$ is a trace such that $AP/\mathcal{T}$ is refined by STOP. Since $AP/\mathcal{T}$ is refinded by STOP no components can be in state 1 after $\mathcal{T}$ because $AP/\mathcal{T}$ must refuse s_i for all i. If all components are in state 2 then any one of them which has an external input, of which there must be at least one, can accept an event from that input and thus $AP/\mathcal{T}$ is not refined by STOP.

Thus there must be at least one component in state 3, say component n_0. Let o_0 be the event corresponding to any blocked output of component n_0 and component $n_1 \neq n_0$ be the destination of output o_0. Component n_1 cannot be in state 1 because if so $AP/\mathcal{T}' \neq$ STOP. Similarly component n_1 cannot be in state 2 because if so $AP/\mathcal{T}'$ cannot refuse o_0. Thus component n_1 must be in state 3.

Similarly there must n_2, n_3, etc such that for all i at least output of component n_{i-1} is blocked by component n_i (and therefore component n_{i-1} must be in state 3). Since there is only a finite number of components there must be at least one cycle of components.

Thus if $AP/\mathcal{T}$ is refined by STOP then it contains a cycle of components that mutualy block each other and it is therefore sufficient to show that $AP/\mathcal{T}$ does not contain such a cycle.

Assume without loss of generaltiy, by renumbering componetns if required, that one of the blocked cycles in $AP/\mathcal{T}$ consists of components $n_0, \ldots, n_{m-1} = 0, \ldots, m-1$ and component n_i has a blocked output to component $n_{i\oplus 1}$ for all i, where $\oplus$ is addition modulo m. Define o_i is a blocked communication event from component i to component $i \oplus 1$ for all i.

The event o_0 must have occurred because $r_1(I(o_0))$ could not refuse it after $\mathcal{T}'$ otherwise. Since o_0 has occurred and component 0 and component wishes to engage in the event again the $s_0 0$ event must have occurred after the o_0 event (otherwise $r_0(O(o_0))$ would refuse the event o_0). Thus $l(\mathcal{T}, o_0) < l(\mathcal{T}, s_0)$. The s_1 event cannot have occurred after the o_0 event, because otherwise $r_1(I(o_0))$, which is part of component 1, would not refuse o_0. Thus $l(\mathcal{T}, s_1) < l(\mathcal{T}, o_0)$. Thus $l(\mathcal{T}, s_1) < l(\mathcal{T}, o_0) < l(\mathcal{T}, s_0)$.

Similarly $l(\mathcal{T}, s_{i\oplus 1}) < l(\mathcal{T}, o_i) < l(\mathcal{T}, s_i)$ for all i. Thus by induction around the cycle of blocked components $l(\mathcal{T}, s_0) < l(\mathcal{T}, s_0)$ and therefore no blocked cycle is possible. Since a trace $\mathcal{T}$ such that $AP/\mathcal{T}$ is refined by stop requires a blocked cycle no such trace can exist. □

This result shows that a simple and implementable component design can guarantee deadlock freedom without imposing significant synchronisation requirements. The `mpkern` parallel programming library, described in the next chapter, is an implementation of the asynchronous program design with the infinitely readable inputs removed and a (separate) shutdown mechanism. Using the `mpkern` parallel programming library eliminates almost all causes of deadlock by the deadlock freedom proof above.

5.4 Summary

A loosely synchronised and flexible model of parallel program design subject to a powerful deadlock freedom result has been described. Only pairwise synchronisation is required and the model can be efficiently implemented. Chapter 9 presents a high performance solution of a "real world" problem built with the implementation described in chapter 6.

The main limitation of asynchronous programs is the need to analyse the data flow, which can be a time consuming process, and high memory consumption if multiple components need a copy of a large object (a shared memory or data parallel environment might allow a single copy to be shared).

Chapter 6

Using The mpkern Library

The asynchronous programs presented in chapter 5 are a flexible design with guaranteed deadlock freedom. Implementing the model in a conventional imperative programming language requires a significant amount of run time support due to its lack of control flow and parallel input and output. Imperative programming languages are languages where the program explicitly tells the computer what to do—unlike functional programming languages like LISP, which evaluate things only if they are needed to evaluate the expression they are asked to compute, for example.

This section describes how to use the `mpkern` library, a lightweight run time system for implementing asynchronous programs with the addition of termination and **removal of the external input sources**, which are not available in many parallel programming environments. An infinitely readable external input is unrealistic for some problems, especially batch problems that are expected to terminate. Many clusters and supercomputers do not allow interactive use.

Fortunately it is easy to void the states where the proof relies on an infinitely readable external input, namely the (otherwise deadlocked) state no active components and no unread messages. An infinite readable external input can be relied upon to do something in this state unless the component it is connected to prevents it, which is impossible.

A program using the library does whatever initialisation it requires and then calls the library function which implements the main loop. The main loop performs all the communication and calls user-provided functions that implement the computation phase of components. This allows these functions to be implemented in an existing programming language which can be compiled using sophisticated optimising compilers.

The library also handles termination and provides various features that some components are likely to require, for example queues that collect an argument from all inputs of a component. These queues are useful for implementing barriers and components which fuse data from multiple sources. A trivial example of such a component would be a component which computes $a - b$, where a and b are its inputs.

The library is designed as a set of core functions and a set of transport specific functions which implement the communication. The differences between the requirements of different communication schemes also put the main loop and shutdown logic in the transport specific functions. The TCP+ transport, which uses TCP connections to communicate between nodes and pointer manipulation to transmit data between two components on the same node, is supplied with the library.

This chapter only describes the API (application programming interface) which is implemented

entirely by the core functions and will remain unchanged if another set of transport specific functions is used instead of the TCP+ transport. An overview of the programming model and implementation in sufficient detail to use the library can be found in this chapter. Further details of the implementation and correctness of the TCP+ transport can be found in chapters 7 and 8, both of which assume some familiarity with the high level description of the implementation in this chapter.

6.1 A library for writing asynchronous programs

A program using the `mpkern` library provides a list of components, connections between them, a set of initial inputs and the component bodies. The body of each component is invoked when a message is delivered to one or more of its inputs. The library provides functions that allow a component to send messages to other components, via any of the component's outputs. The bodies of the destination components will be instantiated as a result.

The `mpkern` library is designed to implement **coarse grained data driven** algorithmic parallelism where the task is split into discrete pieces that process their inputs and pass their outputs to other components. Command line arguments are easily handled and the resulting programs are self contained—there is no need for a command like `mpirun`, which might require you to choose counter intuitive switches to avoid those that the launch program "understands".

The main features of the library include

- built in buffer management.
- formally proved almost complete deadlock freedom. The only possible cause of deadlock is no active components and no messages in transit, which is easily avoided.
- support for per node usage statistics.
- efficient use of network bandwidth.
- compatibility with tools for debugging sequential programs.
- no restrictions on how the components are connected.
- program elements that can be composed simply.

Local analysis suffices to prove deadlock freedom. The only difference from the asynchronous programs analysed above is the absence of external and always infinitely readable inputs and support for (possibly violent) termination, the details of which can be found in the `mpkern` internals documentation. Thus the only possible deadlock scenario is a situation where no component can fire and no messages are in transit, which is easily avoided. The proof in chaper 5 eliminated this scenario by insisting on at least one infinitely readable input. In practice this scenario is easy to avoid.

6.2 Data driven programs

A program based on the library is expected to do the bulk of its computation in data driven components. A component fires when data for it becomes available and it has no incomplete outputs. A special injection operation makes data available from one or more sources initially

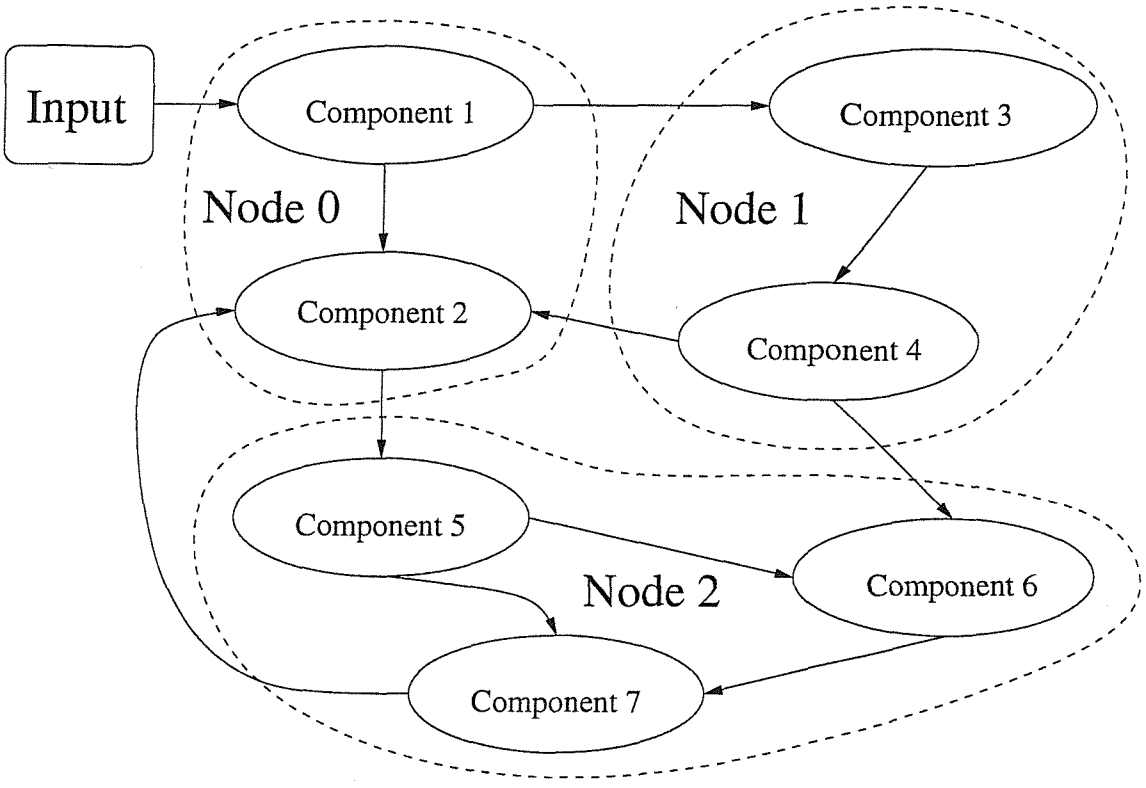


Figure 6.1: Conceptual picture of a mpkern program

before any component has fired. After this the execution is completely data driven. There is **no** conventional control flow between components.

Programs using the library specify how the components are distributed, the connections between components and one or more initial inputs. There is no way a component can send a message, do some processing and then check for a reply. Furthermore, no actual communication will occur until the component has returned to the library. Instead the reply must either be received in a new instance of the same or another component. This style of programming can, at least theoretically, make efficient use of the available parallelism by starting to process data as soon as it is available, avoiding assumptions about the timing of a parallel system.

A seven component program might be distributed on three nodes as shown in figure 6.1; in the figure components 1 and 2 run on node 0, components 3 and 4 run on node 1, and components 5, 6 and 7 run on node 2. Data is read from the input, enabling component 1 to fire. Component 1 may communicate with no components, component 2, component 3, or both as a result. Any component that component 1 communicates with will fire as a result and may cause further components to fire by sending them a message. Any component may perform output using the usual system facilities.

The implementation in the `mpkern` library is markedly different from the model presented above and is illustrated in figure 6.2. Each node is a separate single threaded process, with the main loop in the library. The main loop on node p', which is part of the library, invokes the body of fireable components which runs on node p', and communicates on their behalf. Any message that has to travel from one node to another must be transmitted via an interprocess communication facility, for example a connected TCP socket.

The `mpkern` library is divided into some core functions (`mpkern` core) and transport functions

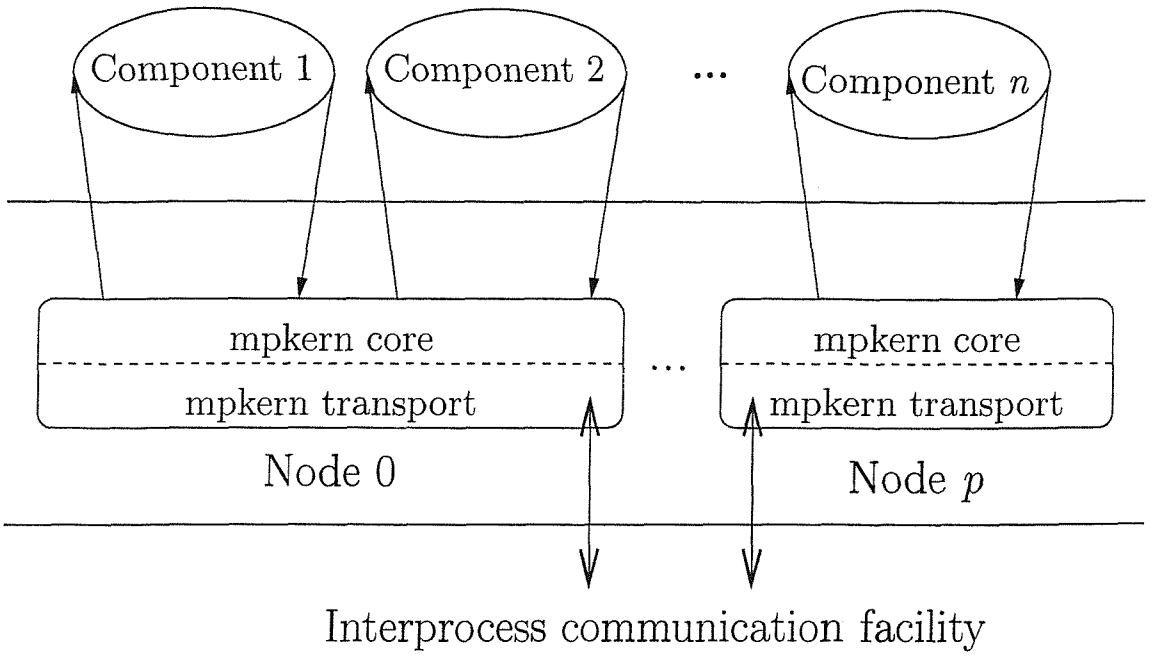


Figure 6.2: Implementation of a `mpkern` program, for n components and $p + 1$ nodes. Single headed arrow show control flow. The main loop is part of the `mpkern` library.

(`mpkern` transport), which implement the message delivery, including sending messages via the interprocess communication facility when required. It is possible to build the library with any set of transport functions that satisfy the requirements.

The processes representing the nodes may be distributed across multiple processors on multiple computers, provided the transport layer supports it. The supplied TCP/IP and in-memory transport, refered to as the TCP+ transport, supports distribution across multiple computers.

The current implementation of the main loop in the TCP+ transport[1] has a communication phase, in which data is sent and received, and then a computation phase which calls the bodies of the fireable components. No component fires more than once in the computation phase of any cycle of the main loop. No communication with other nodes takes place until all the components in the list of fireable components have been fired. This allows the library to aggregate all the messages for each remote node, which makes more efficient use of the network bandwidth and uses less CPU time than sending the messages individually, especially if many small messages are sent.

6.3 Distributing the load

The TCP+ transport implements two transport mechanisms. The in memory transport only works within a node, because it is based on pointer manipulation. Across node boundaries there is no shared memory, so a simple protocol over a TCP connection is used instead.

Groups of components which transfer a lot of data between themselves should share a node provided the serialisation involved is acceptable. The Hartree-Fock application has data aggregation components and components that processes the aggregated data which reside on the same node.

[1]The main loop is transport specific, to simplify handling the differing requirements of different transport mechanisms.

A set of components that do not have large communication requirements, but do use a lot of processor time and could operate in parallel should be spread across several nodes to avoid serialisation[2].

6.4 Timing

Due to the separate communication phase and aggregation of messages, the latency and bandwidth experienced by any given message is a complex function of many factors that are hard to predict for a complex program. The only guarantees on timing are

- Messages are received singly and completely.
- Messages will always be successfully delivered.
- Messages sent over a channel are recieved once and in the order they were sent.

Otherwise the timing may be arbitrary, so if α is sent via A and then β is sent via B five minutes later, then β may arrive before α. The library provides support for various types of message storage queues to simplify correctly handling pathological timing.

The library has a verbosity level, which controls how much information about the firing of components and other events is reported. Higher verbosity levels report everything reported by lower levels and some additional information. Verbosity level 10 and above report detailed information about the library internals, which is only useful for debugging the library.

The following verbosity levels are likely to be useful for debugging programs. All verbosity levels above 0 can generate a large volume of output, which might affect the timing, and are not recommended for large instances.

- verbosity level 0 is the default and prints no messages.
- verbosity level 1 adds when components fire.
- verbosity level 2 adds when components fire return.
- verbosity level 3 adds the size and destination of messages sent and received.
- verbosity level 5 adds the state of queues used to gather complete sets of arguments.
- verbosity level 9 shows when all of a message for another node has been accepted by the interprocess communication facility.

The time on a node is zero when the node passes through the global barrier that prevents entry to the main loop until all the data connections have been established. Unfortunately time is not exactly the same on all nodes and thus times are not directly comparable. The timer resolution is nominally microseconds of wall time[3] but may be coarser in reality.

[2]Serialisation might be better than sharing a processor because the latter also uses processor time for context switching.

[3]time as measured by a clock on the wall, which might be different from CPU time.

6.5 The main program

The public interface of the `mpkern` library is declared in the header file `mpkern.h`. The file declares the public data structures and public functions, both of which all begin with `mp_`. Programs that wish to ensure compatibility with future versions of the library should avoid type and function names beginning with `mp_` and `__mp__`. Functions and global variable names beginning with `__mp__` are used for library internals used in more than one source file, for example the local node number, which is needed only by the library's internal logic—how a correct program is distributed only affects its performance, not correctness.

Many of the functions provided return `void *` pointers. These point to data structures that programs using the library should not attempt to "understand" or manipulate, except by using the library functions provided[4]. The details of these structures might change in arbitrary ways between different versions of the library.

The library points to inputs using `void *` pointers, for the same reason—the data sent is a matter for the program and not something the library should attempt to interpret. Components running somewhere on a homogeneous cluster should not have the library insist on converting the data to and from an external format, which might have a significant impact on the overall performance.

A program based on the library is structured as

1. `#include <mpkern.h>`
2. `mp_maxprocs` *(optional)*
3. `mp_init`
4. *optional program specific initialisation*
5. `mp_inject`
6. `mp_mainloop`

Once the program calls `mp_mainloop` the rest of the computation is executed by component bodies. All components on all nodes are terminated, violently if need be, when any component calls `exit`(3). The list of components, which nodes they run on, and the connections between them are usually fixed in advance.

The way components are distributed across the nodes only affects the timing and performance of the program, provided it is sufficiently robust to cope with any timing that satisfies the guarantees above. Thus a program can be debugged using existing tools, for example source level debuggers, by setting the number of nodes to 1.

Each component has a globally unique name, a body, or work function, which it may share with other components, an assigned node number, and a pointer for the component's private data. The body processes the component's inputs and computes outputs. A component can choose whether or not to use all of its outputs individually every cycle.

Components communicate via *connections*, which connect exactly one source component output to exactly one destination component input. Each connection has a unique name, a source component name, a source index, a destination component name, and a destination index and size. A variable of size 0 accommodates variable size messages, including empty messages (which

[4]hence the use of pointers to `void`.

have a size of 0). The source component name `nowhere` represents external input sources, and is useful for injecting problem parameters and other initial stimuli.

6.6 Startup

`mp_maxprocs`(**int** *nprocs*) sets the maximum number of nodes to *nprocs*, which is initially the largest possible integer (about 2.1×10^9 on 32 bit systems). The number of nodes used is the number of nodes listed in the list of nodes or the maximum number of nodes, whichever is the smaller.

`mp_init`(**int** ***argc*, **const char** ****argv*) starts the remote processes, using a remote shell command. The arguments `argc` and `argv` are adjusted to remove the effect of any extra arguments added by the library, which the library uses to identify remote nodes started by `mp_init`.

A program that wants n nodes given only $m < n$ nodes runs all the processes on node $n' \geq m$ on node $n' \bmod m$. The component and connection list can, at least in principle, be constructed after calling `mp_init` but before calling `mp_mainloop`, which allows it to take the number of nodes into account.

`mp_inject`(**const mp_var** **var*, **void** **data*, **size_t** *size*) injects a copy of the data at *data* with size *size* into the destination of the connection *var*. This only works if it is called before `mp_mainloop`.

The requirement for at least one input before a component can fire makes at least one injected value mandatory. Some concurrency control devices, for example a barrier component that prevents data reaching a compute component until it has notified the barrier that it can accept it, require an initial injection.

After `mp_mainloop` has been called the library invokes the component bodies until a component calls `exit`(3). The control never returns to the function that called `mp_mainloop`. Deadlock is only possible if there are no messages in transit.

6.7 Specifying the connection graph

The components of a program and their connections are described as a directed graph. The communication graph might not be acyclic. Each component corresponds to a node. A communication channel with source component a and destination component b is an arc from a to b. The implementation depends on the underlying transport. The components are specified as a list of `mp_component` structures, terminated by a `mp_component` with the name set to NULL. The structure is defined as

```
typedef struct
{
    const char *name; /* Name */
    int proc; /* Processor # */
    /* The body parameters are
     * Parameters:
       private: For the body's own use.
       handle:  For passing to gater_allargs if used.
       ina pointer to n input, either NULL or pointer to passed data
     * Returns:
```

```
      pointer to new private data.
     */
    void *(*body)(void *private, void *handle, void **in); /* Body */
    void *data;
} mp_component;
```

The name element declares a name, different from the names of all other components, proc the processor number and body is a pointer to a function that implements the processing phase of the component. The data member is an initial pointer to void that the body can use to store persistent state. The library does not interpret the private data or content of messages sent by one component to another.

The connections are specified by a list of mp_var structures, terminated by a mp_var with name, source component name, and destination component name NULL. The structure is defined as

```
typedef struct
{
    const char *name; /* Name */
    size_t size; /* Size */
    const char *from; /* Sending component name */
    int from_idx; /* Sending index */
    const char *to; /* Recipeint component name */
    int to_idx; /* Recipient index */
} mp_var;
```

The name element contains a name, different from all other mp_vars, that is used by the library in error and debugging messages. Setting the *name* component of mp_var to NULL is unsafe. The *size* is a fixed message size or 0, which allows variable size messages. Fixed size messages are a little more efficient than variable size messages.

The *from* and *from_idx* give a source component name and output index number for the source component. The special component name nowhere corresponds to inputs with no source. The *from_idx* of connections from nowhere is ignored. Program parameters can be injected into connections from nowhere to fire an initialisation component.

The *to* and *to_idx* specify a destination component and input index, which must correspond to an existing component. There are no special destination names.

A component with n inputs and m outputs supports input index numbers from 0 to $n-1$ and output indices from 0 to $m-1$. Index numbers are better, and computationally more efficient, than names because they make some common jobs much easier, for example distributing n generated data sets to n components that perform further processing on their piece (in parallel, if they all reside on different nodes).

6.8 Component body invocation

A component body is a function provided by the programmer and matches the C prototype **void** *foo(**void** **state*, **void** **handle*, **void** ***in*).

state is a pointer that the function can use to store component specific state information. *handle* points to an opaque internal library data structure, referred to as a component handle

below, which is used by many library functions. *in* points to a list of pointers to input data. If *and only if* the nth argument is supplied then $*(\texttt{in}+n)$ is not NULL and points to the input data, even if the size of the supplied data is 0. The library does not process or attempt to interpret the message data.

The value that the body returns is used as the value of `state` with which the component body will be called when the component fires again. `state` is a per component value.

6.9 Message information

If n is a used input ($*(\texttt{in}+n)$ is not NULL) then `mp_msg_size`(**void** **handle*, **int** n) returns the size of the message, where *handle* is the opaque handle passed to the component and n is the input number. The result is undefined if an input n has not been used.

`mp_freemsg`(**void** **handle*, **int** n) frees the storage associated with input number n. This call should not be used if the message might be referenced after the call, for example storing it on an argument queue (described in subsection 6.13.1).

6.10 Sending a message

A component that wishes to communicate over a shared variable does this by calling **int** `mp_output` (**void** **handle*, **int** *index*, **void** **data*, **size_t** *size*). *handle* is the opaque pointer passed to the body, *index* the index number of output shared variable, *data* a pointer to the data, and *size* the size of data. The library will transmit the message without processing it or attempting to interpret its content. Only one message per invocation can be sent via a single shared variable.

Sending a message of the wrong size, via a fixed size shared variable, prints a message on the standard error and does not send anything. There is no compulsion on the transport to transmit the message immediately. For example, the main loop of the TCP/IP and in-memory transport has distinct firing and communication phases—any communication to a component on another node will be delayed until the next communication phase and no component will fire more than once in each firing phase.

6.11 Input and Output

No input and output functions are provided. Components can use the standard error and standard output for console output and read and write files as usual (saving some state might be required). Components should avoid using the standard input, because it is used by the library internally.

Operations that block for a significant amount of time should be avoided. The local components cannot communicate or otherwise use the time a component is blocked. This might cause the operating system's buffers to fill and therefore delay components on other nodes, for example by preventing the library from communicating on their behalf.

6.12 Shutdown

Shutdown occurs when the `exit`(3) function is called on any node. The other nodes are terminated immediately, violently if need be. Any processing that is happening on a node when a shutdown is

Figure 6.3: Two possible states of an argument queue

initiated will be terminated immediately. Per node usage statistics can be generated by arranging for a call to `mp_usage_report`, which protects itself from interruption, when `exit`(3) is invoked.

6.13 Queues

Components that fuse data from multiple sources frequently need to temporarily store messages for later processing. The library provides 3 varieties of queues to simplify implementing this.

- *Argument queues* collect arguments from multiple sources into complete sets.
- *Serial number queues* handle data with associated serial numbers, which can be used to refer to the data.
- *Message queues* store messages at the request of the application.

6.13.1 Argument queues

Components that implement computations that require a value from *all* their inputs cannot assume that all the values will be supplied when their body is invoked. It is possible, and in some cases probable, that the body will be invoked several times with several sets of supplied inputs, which together supply at least one value for all the inputs.

Concurrency control barriers can be implemented using argument queues and an argument that indicates that some processing components can accept a new data set. Connections used for this purpose are likely to require a value injected into them to prevent the first data set being stopped at the barrier.

Argument queues collect all the first messages, all the second messages, etc into sets that can be processed together. Messages without a corresponding message from all the other inputs are stored for later processing, when corresponding messages from the other sources are available.

The shaded area in figure 6.3(a) is implemented by the argument queue. The dashed box is a separate argument vector which contains no values. Component x has sent two values, a and b in that order. Component y has sent a single value c.

If component z sends a value, d, then all arguments will have a value, so the first value of every queue is removed from the queue and placed in the argument vector. This results in the state in figure 6.3(b). Further arguments from each components x, y and z will be processed in the order sent when there is an argument from each component available.

An argument queue is constructed by `mp_new_argqueue`(**void** *h, **void** *q). h is the handle passed to the component, and is stored in the queue. If q is NULL a new queue is allocated an (opaque) pointer to its state returned, otherwise the value of q is returned. This is intended to allow code lines of

```
state->qp=new_argqueue(handle, state->qp)
```

where `state->qp` is initially NULL and subsequently points to the queue. Argument queues cannot be constructed outside components because the data pointed by h is not allocated until `mp_mainloop` has been called.

Input data is registered with an argument queue by `mp_gather_allargs`(**void** *q, **void** ****in), which returns either NULL or a **void** ** pointer to a vector of values for all the arguments. q is a queue and *in* the inputs. If *in* has been manipulated or any of the messages have been freed with `mp_freemsg` the behaviour is undefined.

If `mp_gather_allargs` returns a non NULL value then `mp_argq_msgsize`(**void** *qp, **int** n), where qp points to the queue, returns the size of the nth component, for $0 \leq n <$ inputs, of the returned vector of pointers. If `mp_gather_allargs` returned NULL the result of calling `mp_argq_msgsize` is undefined.

6.13.2 Serial number queues

Serial number queues are useful for referencing large data sets shared between several invocations of the body of a component, which come from another source. Requests can include a serial number instead of the data, thus saving complexity and bandwidth, which can be used to find the appropriate data. Serial numbers are used for this purpose in the `beta_worker` component of the task parallel Hartree-Fock program discussed in chapter 9.

A serial number in this context is a non-negative 32 bit integer, between 0 and $2^{32} - 1$, with $a < b$ if $a = b + n \bmod 2^{32}$ for some $n \leq 2^{31}$. This definition of $<$ allows a larger serial number to be generated by adding a small number to an existing one, without special handling of the case when integer arithmetic overflows.

The function `mp_new_serialq`(**void** **handle*, **const char** **name*) creates a new serial number queue and returns a **void** * pointer which can be used to refer to the queue. The *handle* is the handle passed to the component and *name* is a name that need not be unique and is used in a number of messages, which are generated at high verbosity levels.

`mp_seek_serial`(**void** **queue*, **unsigned int** *serial*) and `mp_find_serial`(**void** **queue*, **int** *serial*) returns either NULL or a pointer to the value in *queue* associated with the serial number *serial*, or NULL if no such data is available. `mp_seek_serial` removes entries with small serial numbers, and `mp_find_serial` leaves outdated entries in place.

`mp_add_serial`(**void** **q*, **int** *idx*, **unsigned int** *serial*) adds inputs *idx* to the serial queue q, with the serial number *serial*.

`mp_filter_serialq`(**void** **q*, **int** (**keep*)(**void** *)) selectively deletes entries from a serial queue. The function pointed to by *keep* is called with each message in the serial queue and the item is removed if, and only if, the function returns 0.

6.13.3 Message queues

Message queues provide general purposes message queues which store messages as directed by the application. Their uses include storing requests for processing that require a data set with a serial number that is not yet available. The `beta_worker` component in task parallel `bertha` (chapter 9) uses a message queue to store jobs that request a basis set, via a serial number, that has not yet been delivered.

`mp_new_msgq(void)` creates a new message queue and returns an opaque pointer to it. The queue is initially empty. `mp_msgq_add(void *q, void *handle, int idx)`, where *handle* is the component's handle, adds input number *idx* to the message queue *q*. `mp_msgq_deltop(void *q)` deletes the first entry of the serial queue *q*. `mp_msgq_empty(void *q)` returns 1 if the message queue *q* is empty and 0 otherwise. `mp_msgq_topsize(void *q)`, where *q* is a non empty message queue, returns the size of the top message and `mp_msgq_topmsg(void *q)` returns a pointer to its content.

6.14 cg: A connection generation language

It is possible to generate the connections using a general purpose programming language, as the sample sort example does in C, the amount of code required indicates that this is impractical for a complex program like the task parallel Hartree-Fock program in chapter 9. Thus a special purpose programming language, called `cg`, which is explicitly designed for creating lists of components and connections is provided. Components and connections are created using `component` and `connect` primitives.

The output of a `cg` language program is the `mp_var` and `mp_comp` arrays for a fixed number of nodes. The `cg` language syntax is primarily, although not exclusively, based on C. The keywords are reserved words and may not be used as function names, variable names or unquoted strings. There are no global variables, references, or pointers but almost all their uses can be implemented using output parameters.

Unlike C the `cg` language is not general purpose because it lacks the ability to read and write files, and many other operations that a general purpose programming environment must provide. Other facilities are restricted to an extent that would be intolerable in a general purpose programming language, for example it is only possible to send output to the standard output. The lack of structured data types also limits the usability of `cg` for general purpose programming.

The main advantages of the language are

- Compact specification of components and connections between them.
- Numbers can be combined with strings without explicit type conversion (this is useful for generating component names like `foo0`, `bar2`, `baz42`, etc).
- Built in checking that the connections specified satisfy the rules, so one does not inadvertently call a component nowhere, use output indices 0 and 2 from a component but not 1, etc.
- There is no need to compute the number of components or connections in the graph.

The primary disadvantages are

- `cg` cannot be used to construct the graph on the basis of an external file, or many other factors accessible in a language like C.
- `cg` output does not produce a program that adapts itself to the number of nodes available.

  A parallel program using the structure generate by `cg` for n processors might suffer severe load imbalance when run on $m < n$ processors. The work of processor $n' \geq m$ is added to the load of processor n' mod m by the library, which could double the load on some processors and but not others (assuming all the program is well balanced on n indentical processors).

- Depending on the C compiler, initialisation might be less flexible than generating the graph in a general purpose programming language.

These differences can be illustrated by comparing `connect.c`, the C code that generates the list of nodes and connections in the sample sort application, and `sort-gen.cg`, a `cg` program for generating the same components and connections. The components and connections of the sample sort application, for 3 nodes, are shown in figure 6.4 on page 51. The `cg` grammar can be found in appendix A.

- `sort-gen.cg` is much shorter (only 61 non-blank lines, where the C version has 221 non blank lines).
- `connect.c` explicitly constructs the component names in strings, which are then duplicated when constructing the component and shared variables data structures. `sort-gen.cg` makes extensive use of adding numbers to strings which does the same thing much more compactly.

  For example, one of the connections in the C version, `connect.c`, is constructed using

```
/* Phase 0 to phase 1: generate to sort */
sprintf(vnam, "dist%d", i);
sprintf(dst, "sort%d", i);
(v+var_idx)->name=strdup(vnam); /* Name */
(v+var_idx)->from="generate"; /* Source is generate */
(v+var_idx)->from_idx=i; /* Source index */
(v+var_idx)->to=strdup(dst); /* Destination in sort<num> */
(v+var_idx)->to_idx=0; /* Input index */
(v+var_idx)->size=0; /* Variable size */
var_idx++;
```

  The `cg` equivalent of this is `connect generate(i+1) to "sort"+i(0) via "dist"+i;`. If a size other than 0 was desired it would be necessary to insert `with size` *size* before the semicolon.

- `connect.c` allows the user to specify a maximum number of nodes on the command line. This cannot be possible with code generated from `sort-gen.cg`.
- `connect.c` computes the number of components and connections between them, which would be error prone in complex cases leading to obscure bugs. `sort-gen.cg` computes neither.
- `connect.c` adapts to the number of processing nodes available. Code generated from `sort-gen.cg` would not do this.
- Using `connect.c` one cannot verify that the connections it specified fit the rules, except by run time testing or mathematical analysis of the code, probably using a formal method. The simple connection structure of sample sort make it fairly easy to see that the structure is correctly generated in this instance.

6.14.1 Overview of a cg program

A `cg` program is composed of functions, which do not nest. Spaces, tabs, and new lines are not significant except inside strings or as separators. Any source text between `/*` and `*/` is a comment, which is lexically equivalent to a space.

The interpreter starts a `cg` program by calling the function `main`, which has no parameters and no arguments. The `cg` programming language includes special syntax for generating components and connections between them. The `cg` interpreter inserts the list of components and list of connections at appropriate points in a template after the `cg` program's main function returns.

A constant is any expression involving only constant terms. The number of nodes is a constant called `nodes`. Thus `2*nodes+1` is a constant number and `"foo"+nodes` is a constant string, for example.

6.14.2 Basic types

`cg` has only 3 types of variables: integers, strings and taps.

- a `num`, which can be called an `int`, is an integer. Numbers can be expressed in any format C understands, preceded by an optional sign, namely
  - 0x followed by hexadecimal digits (0 to 9 and a to f).
  - 0 followed by octal digits (0 to 7)
  - Decimal digits (0 to 9) not starting with a leading 0.
- a `string` is a null terminated sequence of characters. A letter followed by an arbitrary number of letters or digits not recognised as a keyword, variable or function name is a string.

  Anything enclosed in double quotes is also a string. Multiple strings in double quotes separated by only white space are concatenated and considered to be a single string[5]. The characters following the symbol \ that ANSI C understands are also understood and represent the same characters.
- a `tap` is an input or output of a component.

A variable is either a single basic type or (a possibly multiple dimensional) array of values of a basic type. There are no pointers, references or structured types, at least some of which are usually available in general purpose programming languages.

6.14.3 Expressions

String expressions

There are two operations defined on strings.

- *string* + *string* concatenates two strings. If either, but not both, is not a string then it is converted to a string.
- *string* `[` $number_1$ `to` $number_2$ `]` is characters $number_1$ to $number_2$ of string. The first character of string is character 0. If $number_1 > number_2$ then the value is the empty string.

[5]This is also true in ANSI C.

Concatenating strings and numbers using + is a convenient way of generating names for groups of components sharing the same body, for example the sort*n* components in the sample sort example described in section 6.16.

6.14.4 Numeric expressions

Relation Operators and Boolean operators

All the comparison operators <, >, <=, >=, == (equal) and != (not equal) are 1 if true and 0 if false. The result is a number for any type of operand. && is a boolean and operator, which stops evaluating the conditions when it encounters one which is false (equal to 0). The || operator is a boolean or operator, which stops evaluating the conditions when it encounters one which is true (not equal to zero). Both the && and || operators evaluate the conditions left to right.

Arithmetic operators

The arithmetic operators are ** (exponentiation), / (division), * (multiplication), % (modulus), + (addition) and - (subtraction). ** has the highest precedence followed by *, / and %, followed by + and -. The fractional part of the result of a division is discarded. All these operators have a slightly higher precedence when applied to a pair of constants. The +, -, * and / work as if the associated left to right, thus `foo-3-2` is the same as `foo-5` and `foo/3/2` is the same as `foo/6`.

The result is sometimes different for expressions like `foo*80/100` and `foo/2*6`. `foo*80/100` is interpreted as `foo*(80/100)`, which integer arithmetic turns into `foo*0`, which is not the excepted result. `foo/2*6` is replaced by `foo*3`, which is different from `(foo/2)*6` using integer arithmetic if `foo` is an odd number. This problem can be solved by using brackets.

Bitwise operators

The bitwise operators are ^ (exclusive or), | (logical or) and & (logical and). All have equal precedence, above comparisons (<, >, etc) and below arithmetic operators (+, -, *, /, etc).

6.14.5 Statements

A statement is either a simple statement or any number of statements enclosed in curly brackets ({...}), which is called a compound statement. Simple statements are variable declarations, assignments, control flow statements, `component`, `connect` or `print` statements, or calls to functions which return no value. Functions which return a value can only be called in a context where their return value can be used, for example an assignment statement.

The `print` statement

The `print` statement prints the value of a single number, string or tap on the standard output. Multiple values can be printed by using string concatenation, for example `print a+" "+b+"\n";` will print `42 69`, and a newline, if `a` is a number set to 42 and `b` is a number set to 69.

Variable declarations

A variable can be declared in any context that a statement is valid. The syntax is *type name dimensions*; where *type* is one the basic types above, *name* is a constant string, different from

`result`, and *dimensions* is zero or more dimensions, which are constant numbers, sometimes related to the constant `nodes`, in square brackets.

Creating components and connections

The `component` *name* `body` *C function name* `on` *node*`;` statement creates a component called *name* with a body of the C callable function *name* that runs on node *node*. A component must exist before taps representing its inputs or outputs and connections to or from it can be created. Connections between components are created by `connect` *source* `to` *destination connect-options*, where *source* and *destination* are connection endpoints. A connection endpoint is *component(index)*, a tap or `nowhere`. `nowhere` is only allowed as a source. *connect-options* is 0 or more of the following

- `size` *number* specifies the message size in bytes
- `type` *string* specifies the name of the data type of the messages, for example a connection that transmits an integer can be specified with `type int`.
- `via` *string* specifies the name of the connection.
- `name` *string* is a synonym for via string.
- `constant` *string* is a name to `#define` to the number of this connection, which is useful for injecting a message into the connection before calling `mp_mainloop`.

Assignments

An assignment has the form *variable=value*. Special syntax applies for taps, which are references to communication endpoints and cannot be manipulated meaningfully. If `tapvar` and `tapvar2` are taps then the allowed forms are

- `tapvar=from` *component(index)*, which makes `tapvar` a reference to output *index* of component *component*. A component named *component* must have been created before this statement.
- `tapvar=to` *component(index)*, which makes tapvar a reference to input *index* of component *component*. A component named *component* must have been created before this statement.
- `tapvar=nowhere` which sets `tapvar` to a connection from the phantom component `nowhere`. Input, for example problem parameters or dummy input to make a component fire, can be injected into connections from `nowhere`.
- `tapvar=tapvar2` which makes `tapvar` a reference to the same endpoint as `tappvar2`. An error occurs if `tapvar2` has not been set to an endpoint before it is assigned to another tap.

6.14.6 Control flow statements

`cg` provides conditionals, `while` loops and `for` loops similar to those found in many programming languages. `cg` has no goto statement, abrupt loop exit (like the break statement in C or java) or exception handling.

Conditionals

Conditionals have the syntax `if` (*number*) `then` $statement_1$ `else` $statement_2$. The `else` $statement_2$ is optional and part of the closest previous `if` statement in otherwise ambiguous cases. If the *number*, which can be an expression, is not 0 then $statement_1$ is executed; otherwise $statement_2$ is executed.

`while` loops

`while` loops have the syntax `while` (*number*) *statement*. The `while` statement first evaluates *number*, which should be an expression, and if it is not 0, executes *statement* and reevaluates the *number*. This cycle continues until *number* becomes 0.

`for` loops

`for` loops execute a statement with a numeric variable set to each value in an arithmetic sequence. The syntax is `for` *variable=number*$_1$ `to` $number_2$ `by` $number_3$ *statement*. All three numbers can be expressions. The `by` $number_3$ is optional and the step size defaults to 1. The loop executes statement with variable set to $number_1$, $number_1 + number_3$, $number_1 + 2 \times number_3$, etc.

The loop terminates when the variable exceeds the $number_2$, if the step size is positive, or becomes smaller than it, if the step size is negative. Due to implementation details a `for` loop with a step of 0 never terminates, but does not generate an error message.

`for` loops evaluate $number_1$, $number_2$ and $number_3$ **once** before entering the loop, and never re-evaluate them. Thus the loop termination test in `n=a to b { b=b-1; }` compares `n` to the value of `b` before the loop body is executed, despite the loop body changing the value of `b`.

Note that `for` loops in `cg` do the test before executing the loop body. Thus for `i=2 to 1 { print "now i="+i+"\n"; }` never executes the `print "now i="+i+"\n";` statement. Similarly `for i=3 to 7 step -2 { print "now i="+i+.\n"; }` never executes the loop body.

6.15 A comparison with MPI

The purpose of this section is to contrast the features of MPI and the `mpkern` library. The facilities provided by MPI are useful for implementing data parallel programs but little use for other types of parallel program, where MPI can act as thick layer preventing access to useful system facilities.

6.15.1 Deadlock freedom

MPI requires the programmer to ensure deadlock freedom and there are many possible causes of deadlock. MPI's poorly documented, or "implementation defined" and often completely undocumented, behaviour in some delicate cases can make avoiding deadlock much more difficult.

If the timing is complex, or poorly understood, then imposing any fixed safe timing could seriously limit performance. The complexity of MPI could make the only reliable alternative, formal deadlock freedom analysis of a specific solution, very complex. Further each design change might require a complete reanalysis. Simplifying the design, and accepting any performance penalty attached, is more common than attempting formal analysis of a complex MPI program.

The `mpkern` library avoids these problems because it restricts the communication pattern to a class which is the subject of a deadlock freedom result. It is usually relatively simple to show

barriers get all the data required, and thus the no messages in transit and no active component scenario is impossible, even in complex examples like the task parallel Hartree-Fock program[6].

6.15.2 Control flow

A programmer can write any style of message passing parallel program in MPI, at least in theory. MPI is designed for data parallel programming and offers convenient functions for performing collective operations. The underlying facilities used by MPI are sometimes more suitable for implementing infrastructure for other types of parallel program than MPI—the `mpkern` library relies on features of `select(2)` that are hard to emulate using MPI, for example.

The `mpkern` library is much more restrictive. The main loop is part of the `mpkern` library and there is no control flow across component boundaries. In particular it is not possible to send a message and wait for a response, instead separate "request" and "response" components must be used.

6.15.3 Synchronisation

A MPI program can use a broadcast followed by a barrier to ensure that all nodes have the same version of some global data, for example the density matrix estimate in the Hartree-Fock problem. All such data will be visible everywhere on a node, so a sophisticated analysis of the data flow is not required.

Components of `mpkern` programs have much more restricted access to data—if a component needs some shared data then it must have its own copy (another component's copy cannot be used instead). It could be argued that this simplifies maintenance because there is no need to worry about data no longer being available when the problem is redistributed.

The `mpkern` library has no support for global barriers because there is no flow control between components. Specific versions of data can be referred to using serial numbers or all the data required can be sent in a single composite message. The Hartree-Fock example uses serial numbers to refer to data with a longer lifetime and composite messages for groups of data with the same lifetime.

6.16 A simple example: Sample sort.

This section uses the library to implement sample sort. The algorithm implemented in this section is slightly modified to handle very small instances correctly. The communication graph of sample sort is acyclic, unlike the Hartree-Fock program presented in chapter 9.

Sample sort is a parallel sorting algorithm. Initially the data is distributed across p nodes. The algorithm in outline is

1. All nodes sort local data
2. All nodes select $p - 1$ splitters and send them to node 0
3. Node 0 selects $p - 1$ global splitters, from the splitters selected in the previous step, and sends them to all nodes

[6]The Hartree-Fock problem is a large scale numerical problem with many "real world" applications

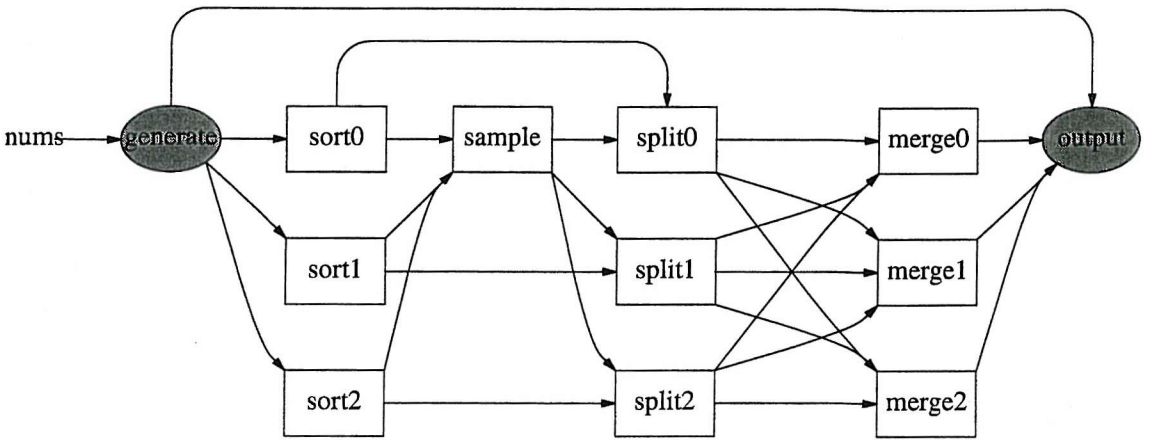


Figure 6.4: Communication structure of sample-sort with 3 nodes

4. All nodes split the local data using the global splitters, sending data between the ith and $i+1$th splitter to node i.

5. All merge their data.

This falls into sort, split and merge components on every node and a global sample selection component. Adding a data generator that feeds the sort component and something to print the data out at the end makes a complete program. The selection of the global splitters, splitting the sorted data and merging all require a complete set of inputs. Argument queues have the behaviour required. The sort components could sort multiple sets of data at once (while one set of data is merged another can be split and a third set locally sorted and sampled).

If there are three nodes available the communication structure looks as shown in figure 6.4. `nums` is an external trigger into which a value is initially injected. The unshaded components implement sample sort. The sort$n$ stages do local sorting and local sampling, sample extracts global splitters, the split$n$ stages do global splitting and the merge$n$ stages do the final merging.

The generate component reads the number passed in via `nums`, generates a random problem of that size and distributes it to the sort components and prints them to the standard output. The output component returns unless it has a value of all the parameters, collected using an argument queue. Once all the values are available the output components checks they are in order, the total number of results is input size, which is sent to it by generate, and prints them on the standard error. After printing the result output terminates the program by calling `exit(3)`.

How the components are distributed is not important for the correctness of the program, but there are some sets of components that can be expected to run concurrently. Other sets of components have dependencies and cannot run concurrently. In the case of the sample sort an obvious partitioning scheme is to run sort$n$, split$n$ and merge$n$ on node n.

The generate, sample and output components could run on any node without major effects on the performance. The code presented runs them on node zero[7].

6.16.1 The main program

The `main` function, which contains the body of the sample sort program (in C) is mildly unusual for a program using the library. A typical, and much simpler, example is found as the main function

[7]This avoids network traffic for output on the standard output and standard error in some configurations.

of the Hartree-Fock program. The extra complexity is largely because the sample sort program is designed for testing the library. The details of the unusual sections have been minimised.

The most prominent unusual features of the sample sort program are that

- The number of nodes and verbosity level can be specified on the command line.
- The list of components and connections is generated to suit the number of nodes, instead of being fixed in advance.

The former makes it possible to see the internals of `mp_init`, provided the debugging level is sufficient. The latter allows a single program to work well on any number of nodes and is feasible due to the simplicity of the way the components are connected.

Generating the component and variable list at run time requires the number of components and connections between them to be computed first. `generate_procs` generates the component list and `generate_vars` generates the list of connections. Every node generates the same list of components and connections independently.

```
beginning of main elided
    node=mp_init(&argc, argv);          /* If remote process argc
                                           and argv adjusted */

    procs=mp_procs();                   /* Get processor count */
    nproc=procs*3+5;                    /* Number of components */
    comp=alloca(nproc*sizeof(mp_component)); /* Allocate components */
    generate_procs(procs, comp);        /* Generate components */

    nv=procs*(procs+5)+3;               /* Number of variables */
    v=alloca(nv*sizeof(mp_var));        /* Allocate variables */
    generate_vars(procs, v);            /* Generate variables */
```

The sample size, which is in a variable called `n` due to a section of `main` not shown here, is injected into the first connection, which is the input of the generate component. The `mp_mainloop` is called with the list of components and nodes, which makes the required connections between nodes and performs the processing using the supplied components. The `mp_mainloop` function should never return.

```
    mp_inject(v, &n, sizeof(n));        /* Inject message */
    mp_mainloop(comp, v);               /* Main loop */
    fputs("Fatal error. mp_mainloop returned\n", stderr); /* Bug trap */
    exit(EXIT_FAILURE);                 /* Die */
}
```

All single and multiple input components are implemented in a similar manner, except for the processing they perform. The processing is not discussed in this section because the focus here is on how a real program uses the library functions. The similarity between the bodies of `mpkern` components and the bodies of input, process and then output components is a design feature of the library—the main difference is that a collection of `mpkern` components deadlock in far fewer situations.

Only the sort$n$ component, all of which have the same body, and sample component are described here. All other components differ from either the sort$n$ or the sample component only in the details of the processing. All of the sort$n$ component body is shown here, in pieces, to illustrate the structure of a complete component body.

6.16.2 A single input component

The sort$n$ and generate components both have a single input. This section describes the sort component. The generate component is similar, except for the processing and different connections, which are used in a similar manner.

The sort$n$ components has a single, variable size, input. This input is definitely supplied when the component body is instantiated because without at least one input the component could not fire. Sorting is `qsort`(3), called with an appropriate comparison function. The sort component does not have any state.

The pointer at `*in` is read and assigned to `buf` (cast to `int *`, which is the message format that generate sends). The number of numbers is computed from the message size. All of the sort$n$ component body is shown here, in pieces, to illustrate the structure of a complete component body. The processing in the sample component is elided.

```
/* Phase 1: Quick sort the sample.
   Input: sample
   Output: sorted buffer, samples
 */
void *sort_sample(void *ign, void *h, void **in)
{
    int i, j, size;
    int *samples, *buf;

    ign=ign;                    /* Avoid compiler warning */
    buf=(int *) (*in);          /* Only one input, thus must be present */
    size=mp_msgsize(h, 0)/sizeof(int); /* Compute #nums */
```

An empty sample (`size` equal to 0) is only possible for very small examples. This code communicates no samples and an empty sorted sample if the sample is empty. Otherwise it attempts to allocate memory for storing the local splitters and exits immediately if no memory is available.

```
    if (size==0)
    {
        /* Normal sample sort does not need size 0 handling here */
        mp_output(h, 0, buf, 0);    /* No data */
        mp_output(h, 1, buf, 0);    /* No samples */
        return NULL;
    }

    if ((samples=xmalloc(sizeof(int)*mp_procs()))==NULL)
        exit(EXIT_FAILURE); /* Die */
```

The next section of code uses the C library function `qsort(3)` to sort the data and extracts regularly spaced local splitters. `numcomp` compares the integers pointed to by two pointers.

```
/* Work */
qsort(buf, size, sizeof(int), numcomp);
for (i=0, j=0; i<mp_procs(); i++, j+=(size/mp_procs()))
{
    samples[i]=buf[j];
}
```

The final section of the body sends the sorted input buffer to the split component on the same node (via output 0) and the samples to the sample component, which selects the global splitters (via output 1). The code works even if fewer than the desired number of splitters could be extracted, which is only possible for very small examples.

```
/* Output results */
mp_output(h, 0, buf, size*sizeof(int));
mp_output(h, 1, samples, i*sizeof(int));

free(samples);              /* Free memory */
return NULL;
```

6.16.3 A multiple input component

Generating the global splitters, splitting, merging and producing the final output components all have multiple inputs and require all of them. The sample component is described here, and implemented by `merge_splitters` which generates the global splitters. All the other multiple input components with one or more outputs are similar—all use argument queues, with their private data holding the opaque pointer to `void` which `mp_new_argqueue` returns.

Computing the global splitters requires samples from all the sort components but is otherwise not history sensitive[8]. The private data pointer is the opaque pointer that `mp_new_argqueue` returns. The initial value of the private data pointer is NULL.

This section allocates a new argument queue, unless one has already been allocated, and uses it to gather a complete set of arguments. The persistent state is the opaque pointer returned by `mp_new_argqueue`.

```
/* Phase 2: Compute global splitters
   Input: samples from everyone else
   Output: set of global splitters
*/
void *merge_splitters(void *d, void *mp_h, void **in)
{
    heap h;
    int i, j, k, tmp, nprocs;
    merge_source *srcs;
```

[8]The results of a history sensitive function depend on some state, which depends on the previous invocations of the function, in addition to the parameters. A mechamism to save the state is required to implement a history sensitive function.

```
    int *splits;
    void **args;

    d=mp_new_argqueue(mp_h, d);
    args=mp_gather_allargs(d, in);
    if (args==NULL)
        return d;
```

`mp_gather_allargs` returns NULL unless an input has been read from all the inputs. Values are stored in the argument queue for later processing if it cannot be processed immediately. If the component does not return here then `d` points to an array of `void *` pointers, none of which are NULL, which point to a list of splitters from all the sort$n$ components.

Merging the splitters is performed by a heap based merge. The details of the merging and selection of the global splitters are not presented here. Finally the splitters are sent to all the local splitting stages. The state pointer is the opaque pointer returned by `mp_new_argqueue` above. `nprocs` is the number of nodes, as returned by `mp_procs`.

```
    for (i=0; i<nprocs; i++)
        mp_output(mp_h, i, splits, (nprocs-1)*sizeof(int));
    return d;
}
```

The output component collects a complete set of inputs using an argument queue in the same manner as `merge_splitters`. Given a complete set of outputs it checks that the outputs are sorted, and that the length of the sorted list is the same as that of the generated list. If the list contains under 400 elements it prints them to the standard output, which is easily retrieved in many batch processing environments. Once this has been done the program is terminated by calling `exit(3)`.

6.16.4 Generating the sample sort connections using cg

The cg language described in section 6.14 it makes much simpler to generate the list of components and connections between them for the sample sort program, described in section 6.16. Extensive use is made of adding numbers to strings and output parameters. This example generates the components and connections shown in figure 6.4, if the number of nodes is set to 3.

The oneproc function generates the sort$p$, split$p$ and merge$p$ components that exist on all components and returns an output tap that returns the final sorted result (the only output of merge$p$). Output parameters are used to return taps for sending the local sample, receiving the global splitters, sending the splitters and receiving the data from the split$p$ stages. The output parameters `source`, `outsamp`, `insamp`, `outsplits` and `insplits` represent the only inputs of the sort$p$ components, an output for sending the local sample, an input for receiving the global splitters, outputs for sending the split$n$ data to the merge$n$ stages, and inputs for receiving data from the split$n$ stages, respectively.

```
/* sort and sample, split and merge */
tap oneproc(in num p, out tap source, out tap outsamp, out tap insamp,
            out tap outsplits[nodes], out tap insplits[nodes])
{
```

```
        /* First make some components */
        component "sort"+p body "sort_sample" on p;
        component "split"+p body "split_data" on p;
        component "merge"+p  body "merge_data" on p;

        /* Connect them togther */
        source=to "sort"+p (0);
        connect "sort"+p(0) to "split"+p(0) via "data"+p;

        /* Export appropiate taps */
        outsamp=from "sort"+p(1); /* An output tap */
        insamp=to "split"+p(1);   /* An input tap */
        num i;
        for i=0 to nodes-1
        {
                outsplits[i]=from "split"+p(i);
                insplits[i]=to "merge"+p(i);
        }

        result=from "merge"+p(0); /* result is the result returned */
}
```

The `sort` function connects together components created by `oneproc`, returning nodes inputs, for input data, and output, for reading the results. The fact that `outspl[i]` is a one dimensional array of size `nodes` (because `outspl` is a `nodes` × `nodes` array) simplifies the function.

```
void sort(in num n, out tap inp[nodes], out tap outp[nodes])
{
        tap insamp, outsamp, outspl[nodes][nodes], inspl[nodes][nodes];

        component "sample" body "merge_splitters" on 0;
        num i;
        for i=0 to n-1
        {
                outp[i]=oneproc(i, inp[i], outsamp, insamp, outspl[i],
                                inspl[i]);
                connect outsamp to sample(i) via "insamp"+i;
                connect sample(i) to insamp via "outsamp"+i;
        }
        num j;

        /* Connect the split and merge stages */
        for i=0 to n-1
        {
                for j=0 to n-1
                        connect outspl[i][j] to inspl[j][i] via "split"+i+"-"+j;
        }
```

```
        /* note: result is not defined for void functions */
}
```

The main function, which is the body of the program, generates a sample sort network using the sort function and connects it to a number generation and result verification component.

```
void main()
{
        tap output[nodes], input[nodes];
        component "generate" body make_testdata on 0;
        sort(nodes, input, output); /* Generate main sort */
        /* Wire it up to test rig */
        component "print" body print_output on 0;
        connect nowhere to generate(0) via "nums" type int constant SIZE;
        connect generate(0) to "print"(nodes) via "sample_size";
        num i;
        for i=0 to nodes-1
        {
                connect generate(i+1) to input[i] via "feed"+i;
                connect output[i] to "print"(i) via "out"+i;
        }
}
```

The generated output for 2 nodes is

```
/* These structures are for 2 nodes */

#define SIZE 5
mp_var mprog_mp_vars[]=
{
    {"out1", 0, "merge1", 0, "print", 1},
    {"feed1", 0, "generate", 2, "sort1", 0},
    {"out0", 0, "merge0", 0, "print", 0},
    {"feed0", 0, "generate", 1, "sort0", 0},
    {"sample_size", 0, "generate", 0, "print", 2},
    {"nums", sizeof(int), "nowhere", 0, "generate", 0},
    {"split1-1", 0, "split1", 1, "merge1", 1},
    {"split1-0", 0, "split1", 0, "merge0", 1},
    {"split0-1", 0, "split0", 1, "merge1", 0},
    {"split0-0", 0, "split0", 0, "merge0", 0},
    {"outsamp1", 0, "sample", 1, "split1", 1},
    {"insamp1", 0, "sort1", 1, "sample", 1},
    {"data1", 0, "sort1", 0, "split1", 0},
    {"outsamp0", 0, "sample", 0, "split0", 1},
    {"insamp0", 0, "sort0", 1, "sample", 0},
    {"data0", 0, "sort0", 0, "split0", 0},
    { NULL, 0, NULL, 0, NULL, 0 }
};

mp_component mprog_mp_comps[]=
{
    {"print", 0, &print_output, NULL},
    {"merge1", 1, &merge_data, NULL},
    {"split1", 1, &split_data, NULL},
    {"sort1", 1, &sort_sample, NULL},
    {"merge0", 0, &merge_data, NULL},
    {"split0", 0, &split_data, NULL},
    {"sort0", 0, &sort_sample, NULL},
    {"sample", 0, &merge_splitters, NULL},
    {"generate", 0, &make_testdata, NULL},
    { NULL, 0, NULL, NULL }
};
```

Passing these values of `mprog_mp_vars` and `mprog_mp_comps` to the `mp_mainloop` function generates a 2 processor instance of the sample sort network. The `SIZE` constant is the connection number into which the input problem size should be injected before calling `mp_mainloop`.

The C connection generation function is designed so that the problem size connection is always the first connection. This avoids the need to calculate the connection number into which the problem size should be injected, which might otherwise depend on the number of nodes. Using fixed connections and components, generated in advance, obviously avoids this problem.

6.17 Summary

A new parallel library was described, briefly compared with MPI and used to implement a simple example. A small special purpose programming language, called cg, is provided for generating the required data structures. The performance of this library is assessed using the Hartree-Fock problem in chapter 9.

Chapter 7

mpkern Library Internals

This section describes the implementation of the mpkern library, which was described in chapter 6. The library is designed to efficiently exploit moderate size clusters of homogeneous systems, largely because of their widespread availability. The aims of the implementation are

- **Correctness:** The implementation should not deadlock, or diverge, for any timing, given any correct program.
- **Efficiency:** The implementation should operate efficiently for both small and large systems and programs.
- **Scalability:** The reliance on centralised control and other unscalable features should be minimised.
- **Flexibility:** It should be easy to add support for another transport mechanism, for example a light weight protocol or low latency networking technology.
- **Simplicity:** The implementation should be as simple as possible consistent with the other goals.
- **Portability:** The implementation should work on a range of systems without any changes. This goal was not that onerous due to the similarities between different unix variants.

7.1 Design

Each node is represented by a single threaded process which covers all the components whose bodies are executed on that node. This is safe because it only affects the timing, which does not matter as shown in the proof in chapter 5, provided the message transmission and reception can happen in parallel. The latter can be implemented using non-blocking I/O, or asynchronous I/O.

Sending a message between two components on different nodes requires the message to be sent from one process to another, which requires some sort of interprocess communication. This requires some sort of underlying interprocess communication facility, for example TCP[28] connections.

To minimise the restrictions on the interprocess communication mechanism, and implementation of communication between components on the same node, the library is divided into core functions and transport dependent functions. Assumptions about the interface to the interprocess communication facility are minimised by putting the main loop in the transport dependent portion of the library. The library core functions implement

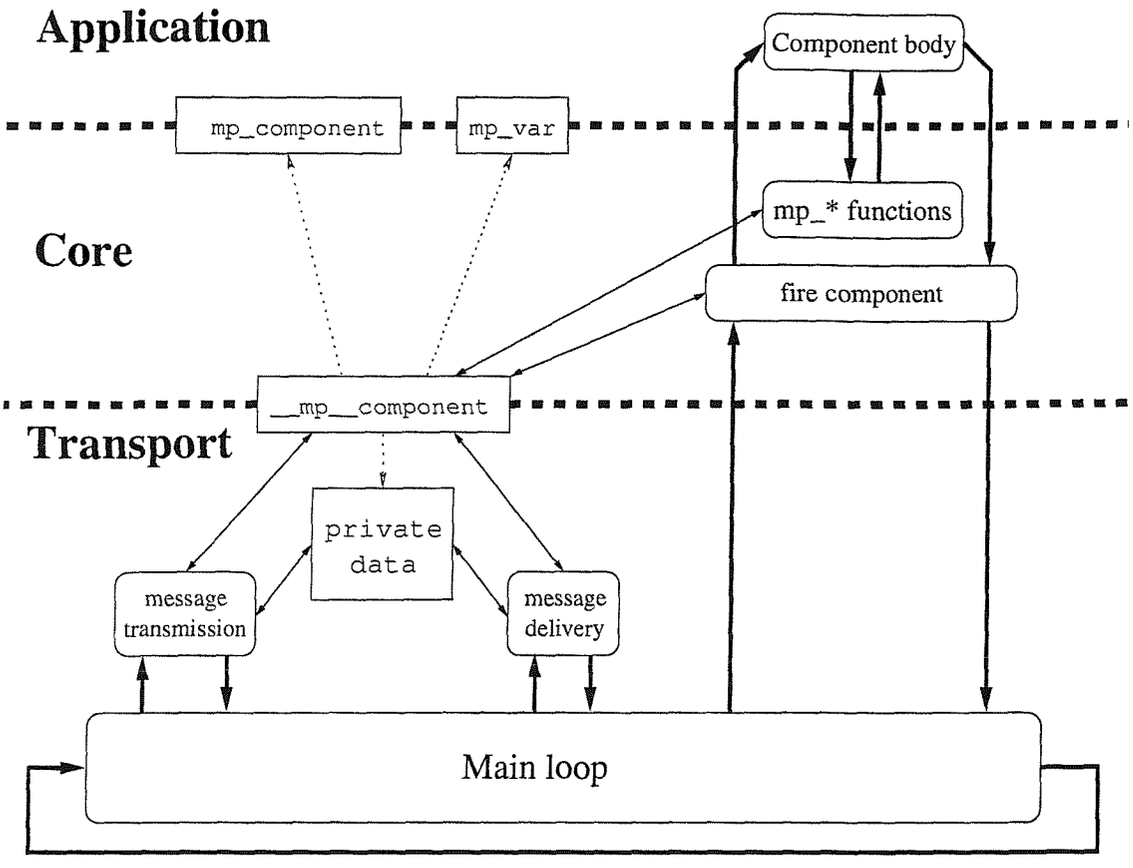


Figure 7.1: Simplified view of the relationships between the major data structures and functions in the mpkern main loop, which is part of the library. The rectangles are data structures and boxes with round corners are important functions or groups of functions.

- the public interface (all the mp_* functions, most of which are documented in chapter 6).
- starting remote processes, using a remote shell command.
- mapping component names to destination numbers during startup.
- building a list of local components, and counting their inputs and outputs.
- injecting the initial inputs into the system.
- invoking a component body.

A simplified view of the main functions, or groups of functions, and data structures in the main loop is shown in figure 7.1. The heavy dashed lines show the divisions between the application, core and transport specific components. The boxes with square corners are data structures and boxes with round corners major functions, or groups of the functions.

The dotted arrows show the direction of pointers. A double headed arrow connecting function F to data structure S indicates that F processes the data structure S. The heavy solid arrows indicate control flow. There is no requirement that the main loop calls the functions in the order suggested, only that it does not deadlock given any set of components that would be deadlock free if executed in parallel, and all messages are delivered in an order consistent with the guarantees given in chapter 6.

Clearly the `__mp__component` structure, which is constructed by a core function before the main loop is entered, is the most important structure. There is one `__mp__component` structure for each local component. All the public functions for use in component bodies, the `mp_*` functions in figure 7.1, invoking a component body and communication all manipulate the mp component structure. The opaque handle passed to a component body is actually a pointer to the component's `__mp__component` structure and `mp_output`, `mp_msgsize`, etc depend on this fact.

The transport delivers a message by adjusting the non-private portion of the `__mp__component` structure to reflect the presence of the message. The `__mp__fire_body` function, which fires a component will determine the presence of the message by examining the `__mp__component` data structure and pass it on to the component body.

When the control returns to the main loop the main loop examines the `__mp__component` to determine which outputs were used and takes transport specific steps to transmit them. Transmission is likely to use some of the private data.

7.2 Why a single thread?

The CSP description has many concurrent components, and requires parallel input and output; the implementation is single threaded. A major reason for both choices is simplicity. A range of designs with multiple threads could have been chosen instead, without sacrificing much portability due to the broad support for POSIX threads.

It would, in principle, be possible to add support for a compute phase and for it to control whether or not the outputs are used to the CSP model, by adding appropriate events. It could then be shown that the two systems are the same when the added events are hidden, using appropriate laws of CSP. The expanded version of the CSP process could then be implemented directly using the same number of threads and a parallel programming toolkit.

Unfortunately the performance of such a design would probably be very poor—the vast number of threads could act as "bacteria" and consume a large amount of CPU time context switching, the process the operating system uses to maintain the illusion that each process has the CPU to itself. The cost of synchronising the threads would also be significant (partly because each system call involves two context changes).

The only solution to the problem of many threads acting as "bacteria" and consuming vast amounts of processor time synchronising and context switching is to reduce the number of threads. One thread per component is fairly easy to implement using existing system facilities and parallel libraries. This design entails n^2 connections where n is the number of components and experiments indicate this leads to poor performance.

Using only 1 thread only affects the timing, which the deadlock freedom proof shows is irrelevant, and reduces the number of connections to $p - 1$ connections on each node, where p is the number of nodes. The amount of context switching and synchronisation requirements are also drastically reduced.

A single thread was chosen to simplify the implementation and maximise performance. Some of the optimisations, for example the aggregation of many small message with the same remote destination and local message delivery by pointer manipulation, would be very difficult to implement if more threads were used instead.

However the requirement to show that the implementation matches the CSP description increases as the distance between the implementation and CSP model it is supposed to implement

increases. Chapter 8 states the most important invariants, and argues that they are maintained using assertional reasoning. These invariants and a progress property are then used to argue that sending a message will cause the destination component to fire with that the message as an input.

That sending a message will causing the destination to fire with the message as an input is sufficient to establish that cycles of components all blocking each other are impossible. Thus the correctness analysis shows enough to prove the deadlock freedom result. The only exceptions is cases in which deadlock would occur without an external input, which is not implemented, as stated in chapter 6. Fortunately these cases are easy to avoid and not guaranteed to be deadlock free by the library.

7.3 The TCP+ transport

The library is supplied with the TCP+ transport, which uses fast pointer manipulation transport between two local components (the in-memory transport) and sends data to components on other nodes via a TCP connection. TCP connections provide a simple reliable bidirectional byte stream connection between two points via a potentially unreliable network, usually an IP (Internet Protocol) network[1]. There is no concept of messages or message boundaries associated with a TCP connection.

7.3.1 Connections

Each node in a `mpkern` program, which corresponds to a process in a unix-like environment, linked with the TCP+ library has two classes of connections. Every node has a single data connection to every other node and a control connection to node 0. Node 0 has control connections to every other node. The control connections are used to communicate before the data connections are set up and implement a centralised barrier during shutdown and before entering the main loop.

The barrier protocol involves every node, except node 0, sending a message to node 0 when it reaches the barrier. When node 0 has reached the barrier and received messages that indicate that all other nodes have reached the barrier, it sends them all a message and passes through the barrier. Nodes other than node 0 pass through the barrier when they receive the message node 0 sent to them.

7.3.2 The main loop

The TCP+ transport's main loop has a separate communication phase, in which it communicates with the other nodes, and a computation phase in which it fires the local fireable components. The communication phase reads messages, or parts of them, from any node and sends messages to writable destination nodes. Components can receive messages, or parts of them, even if they are unable to fire because they have an unsent message.

The TCP+ transport solves the lack of message boundaries by interpreting the data from a TCP connection as a stream of destination numbers followed by message data. The destination number identifies the destination of a message and the connection. If the connection in use carries variable length messages the size of the message is sent as part of the message data.

The sharing of a single TCP connection between all the connections from a component on processor i to processor j makes it difficult to control the destination to which messages are sent.

[1] IP packets may be delivered in a different order from which they are sent, more than once or not at all.

Instead of implementing any control of this the TCP+ transport allows multiple messages to be delivered to all destinations and ensures the component fires once due to every queued message. The presence of buffering does not undermine the deadlock freedom result, because buffering could be implemented by replacing a direct connection with one via an appropriate asynchronous component.

All connections are non-blocking, eliminating the need to worry about blocking during transmission or reception. This decision makes it possible to receive one or more messages in fragments. The interpretation of a byte stream is not affected by how it is fragmented. The transmission and reception processes maintain state and use it to handle fragmented messages. The amount of an input that has been received and an output that has been sent is stored in the elements of the `__mp__component`'s message queues.

The transport specific data attached to each `__mp__component` maintains a list of components with an unsent message, a list of components to fire, and a list of components to fire in the next round. The private data, and balanced tree mapping input indices to an `__mp__component`, suffice to maintain these lists without enumerating the local components.

The initial list of components to fire is generated by examining all the local components, which might be slow if there are many local components. This is acceptable as it only has to be done once.

7.3.3 Shutdown

Shutdown occurs when any node receives a fatal signal, presumably due to a bug, or calls `exit`(3), which invokes special shutdown code attached using `atexit`(3). The shutdown code terminates all the other nodes, using violence if required.

If this happens on a slave node then the slave node closes all data connections and then sends the master node urgent data over the control connection, which interrupts whatever the master node is doing and delivers `SIGURG` to the master node. When the master node catches `SIGURG` it closes all data connections and then sends urgent data to all slaves before exiting. When the slaves catch `SIGURG` they close all data connections and then exit.

Shutdown due to a signal or `exit`(3) on the master node is the same, except that the first step, sending urgent data to the master node, is omitted. These events happen without any error handling because closing all data connections might cause some nodes to have initiated shutdown, so that they may have terminated by the time the master node attempts to send them urgent data.

The violent nature of the shutdown process makes it easy to limit the program's wall clock running time, on systems similar to unix, by calling `exit`(3) at a given time in the future, for example using the `alarm`(2) system call and a signal handler that calls `exit`(3).

7.4 CG implementation

A `cg` program is excuted in three phaes; first the program is translated into a stack machine langauge, then the stack machine language is interpreted and finally the generated components and conections are inserted into a template. Inserting the results of processing the input is also used by the `yacc` parser generator.

A stack machine was chosen as the intermediate format because stack machines are easy to implement and relatively efficient. Performance is not critical in `cg` because `cg` is not involved in

the execution of a parallel program.

The `component` and `connect` primitives add a component or connection, respectively, to a list which is used in the third phase when the list of components and connections is inserted into the template. It would be significantly more complex to generate a C from `cg` instead of inserting its results into a template.

7.4.1 The target machine

The target machine is a stack machine with local variables and a global stack. Arguments and results are passed by value via the stack. There are five possible types of value on the stack and fifteen instructions, which are shown in figure 7.2. The available operators are shown in figure 7.3.

Each item on the stack is one of the following

- A dimension
- An integer
- A string
- A communication endpoint (tap)
- A variable reference

Each instruction is one of the following:

- Call an operator (1 instruction).
- Push a basic data type, that is a number, string or tap (3 instructions).
- Duplicate the element n from the top of the stack and push it on the top of the stack (1 instruction)
- Move the element n from the top of the stack to the top of the stack (1 instruction)
- Pull a number from the stack and jump if it is, or is not, 0 (2 instructions).
- Perform an unconditional relative jump (1 instruction)
- Enter or leave a group of instructions some of which may be skipped (2 instructions).
- Call a function and terminate a function (2 instructions)
- Change the line number or function name (2 instructions).

All functions are expected to start by setting the function name and finish with an `end` instruction. The call an operator function is used for all computation, comparison, creatation of variables, and reading and writing variables. All variables are automatically freed when the function in which they are defined returns.

Instruction	Stack level change	Parameter	Description
operator	Variable	Operator	Call an operator
number	+1	Number	Push a number
string	+1	String	Push a string
tap	+1	Tap	Push a tap
dup	+1	Number	Duplicate items
topmove	0	Number	Move item to top of
call	Variable	Number	Call a function
zjump	−1	Number	Pull n and jump if $n = 0$
nzjump	−1	Number	Pull n and jump if $n \neq 0$
jump	0	Number	Unconditional
enter	0	None	Enter group
leave	0	None	Leave group
end	0	None	Terminate
linenum	0	Nunber	Set line number
funcnam	0	String	Set function name

Figure 7.2: Stack machine instructions

7.4.2 Variables

Variables are created and referenced and their values read and written, using operators. The valid variable names in any context are stored in an ordered splay tree. Calling a function creates a new empty context; returning destroys the current context and restores the previous context. A function's prologue creates the local variables and pulls input parameter values off the stack and its epilogue pushes output values on the stack.

Reading and writing a variable takes place in two stages. Reading the value of a variable first pushes the details on the stack and converts this into a variable reference. The second stage reads or writes the variable.

Defining and resolving variables

The state of the top of the stack (with no dimensions if $n = 0$) when the `addvar` operator is called, is:

<dimension 1> ... <dimension n> n <name> <type number>

All values are numbers except <name>, which is a string. The operator pulls the type number, variable name and dimensions off the stack. It adds an entry to the table of local variables recording this information and allocates storage for the values. Input parameters might have some dimensions set to 0, which will result in the assignment operator setting that dimension from the parameter value.

The top of the stack when resolving a variable when the `vresolv` operator is called, with no indices if $n = 0$, is:

<index$_1$> ... <index$_n$> n <name>

All values are numbers except <name>, which is a string. The name is looked up in the table

Operator	Pulls	Pushes	Overall	Description
nop	0	0	0	Do nothing.
blackhole	Variable	0	≤ -1	Pull n and pop n elements.
print	1	0	-1	Print top element on stack
add	2	1	-1	Add
sub	2	1	-1	Subtract mul 2 1 -1 Multiply
div	2	1	-1	Divide mod 2 1 -1 Compute modulus
pow	2	1	-1	Exponentiate
neg	1	1	0	Unary minus land 2 1 -1 Bitwise and
lor	2	1	-1	Bitwise or
lxor	2	1	-1	Bitwise xor
lt	2	1	-1	Less than lteq 2 1 -1 Less or equal
gt	2	1	-1	Greater than gteq 2 1 -1 Greater than or equal
eq	2	1	-1	Equal
neq	2	1	-1	Not equal
ssand	1	0 or 1	0 or $+1$	Short circuit or
ssor	1	0 or 1	0 or $+1$	Short circuit and
concat	2	1	-1	Concatenate two strings
substr	3	1	-1	Extract substring
stringcmp	2	1	-1	Compare strings
strindex	2	1	-1	Extract character from string
strlen	1	1	0	String length
string	1	1	0	Convert number to string
tapstr	1	1	0	Convert tap to string
tapify	3	1	0	Convert direction, number and string to tap
comp	4	0	-4	Create component
connect	5	0	-5	Create connection
addvar	Variable	0	≤ -3	Create variable
vresolv	Variable	1	≤ -1	Resolve variable reference
value	1	1	0	Fetch variable value
assign	2	0	-2	Set variable value

Figure 7.3: Stack machine operators

of local variables and an error occurs if the top of the stack is not a string or a valid variable name. Both conditions should be impossible, because the former indicates a serious bug in the code generator and the latter a bug in the lexical analyser, which distingishes between key words, variable names and other strings during the compilation phase.

The result contains

- The number of dimensions supplied, n.
- The offset $\prod_{i=1}^{n} \left(\text{index}_i \times \prod_{j=1}^{i-1} \text{dimension}(\text{<name>}, j)\right)$
- A pointer to the variable's information.

If too many indices are supplied or if any index is less than 0 or greater than or equal to the size of the corresponding dimension, an error occurs.

Reading and writing values

Reading a value is simpler than assignment because it cannot affect the dimensions of the variable. Multiplication of the offset by the remaining index values is performed by the `value` and `assign` operators, to simplify handling of arrays with one or more unspecified dimensions.

The `value` operator multiplies together any remaining dimensions to compute the number of values in the result. The offset is computed as the offset computed above multipled by the number of values. `ensure_space` is called to ensure that enough space is available on the stack and then the values are pushed onto the stack, with the value in the highest location pushed first. The dimensions of the result, if any, are then pushed, the dimension with the highest index first. The resulting stack state is

$$\text{<value}_n\text{>} \ldots \text{<value}_1\text{>} \text{<dimension}_k\text{>} \ldots \text{<dimension}_1\text{>}$$

A scalar is represented by a single value and no dimensions.

The `assign` operator might have to allocate memory and adjust unspecified dimensions of the variable. The value must have the right number of dimensions; otherwise the compilation phase reports a type error. This requires the number of values calculation to be able to adjust the variable's unspecified dimensions and compute the appropriate amount of space. It is simpler to calculate separately the number of values, `nv`, and the overall size, if they are needed.

```
varv = 1
nv = 1
nd = variable reference's dimension + 1
offset = variable refenece's offset
flag = FALSE
while (nd ≤ variable's dimensions)
    dim = pop dimension
    offset = offset × dim
    nv = nv × dim
    if (variable's nth dimension = 0)
        variable's nth dimension = dim
        flag = TRUE
    if (variable's nth dimension ≠ dim)
        Run time error
    nd = nd + 1
if (flag)
    nd = 0
    varv = 1
    while (nd ≤ variable's dimensions)
        varv = varv × variable's nth dimension
    Allocate space for varv items
    Initialise varv items
```

This works by calculating the number of values and offset using the value as the accurate information, updating the variable if necessary. Incompatible values are prevented by type checking in the compiler and the test to ensure that the dimensions match, which will fail if a value tries to update a non-zero dimension. If a variable's dimensions are altered the flag is set, the total size calculated and data allocated.

The present implementation only adjusts dimensions of input parameters, because unspecified dimensions are only allowed for input parameters. The unspecified dimensions of parameters are represented by dimensions of size 0.

7.4.3 Control flow

All control flow statements are implemented with conditional and unconditional jump instructions. The comparisons are operators which either push 0 or 1, depending on whether the comparison was true or false.

Conditionals

There are two types of conditionals: those with an else clause and those without. The implementation of both is straightforward. The code generated is

```
        compute test
        conditional jump to no
        code if true
        jump to out
no      code if false
out
```

If the `else` clause is empty then there is no code if false and the jump to out at the end of the code if true is eliminated. There is no attempt to predict the most probable outcome of the test or to analyse how the result of a test affects the results of other tests.

Loops

Loops are implemented as

```
        initialisation
top     compute test
        conditional jump to out
        loop body
        jump to top
out     cleanup
```

`while` have empty initialisation and clean up sections. `for` loops are more complicated because the variable, bounds and step size must be computed once at the beginning of the loop. `for` loops with a specified step are implemented slightly differently from `for` loops without a specified step size. Whether or not a step size is specified, `for` loops terminate when $(\texttt{value} - \texttt{endvalue}) \times \texttt{step}$ is greater than 0. If the step size is 1 this is simplified to $\texttt{value} > \texttt{endvalue}$.

The initialisation step of `for` loops without a step size assigns the loop variable to the starting value and leaves the stack containing `reference to variable`, `end value`. The test duplicates both values and then invokes the > operator, and exits the loop if the result is not zero. The clean up section pops the variable reference and end value off the stack.

`for` loops with a specified step size are similar except that the stack contains `variable reference`, `end value`, `step size` after the initialisation and more duplication is required. The clean up step pops all three items stored on the stack when the loop terminates.

7.5 Summary

An overview of the design of the `mpkern` library and the implementation of the `cg` programming language is described. Programs based on the `mpkern` library have 3 distinct layers: the application layer, the library core and the transport layer (the last two layers are part of the library). The library core provides little except a public interface to the transport layer.

Chapter 8

Correctness of mpkern library

The correctness of the library requires a more detailed analysis of the library than was presented in the preceding chapter. This chapter concentrates on the implementation of the main loop in the TCP+ transport, because this is where most of the opportunities for deadlock and violations of the design rules arise. The main loop is responsible for only firing components when they have no unfinished output and have at least one readable input.

This chapter analyses a single instance of a single threaded process, with no shared data. A parallel program based on the `mpkern` library is a network of these processes, usually one per processor. Thus atomicity is not an issue because no data will change or be examined unexpectedly. This simplifies the design and avoids the CPU time consumed by context switching.

The focus will be on more complex requirements, for example that if a message is sent to a component it will fire with that message as one of the used inputs. The simpler facts can be proved by induction. For example, the accuracy of the queued message counter can be established by noting that the operations that add and remove queued messages adjust the counter as required, and it is initially correct. Queued messages are covered in more detail below.

The argument presented here uses assertional reasoning, which asserts things about the state at particular points in the algorithm. A number of invariants, facts which the code can **always** rely on, are used to simplify the analysis.

The state is required to satisfy the invariants before and after **every** statement, and many operations inside the main loop rely on these properties for their correctness. For example the correctness of "deliver message to component d", which is a single statement within the main loop, relies upon all components being on at most one of *outlist*, *firelist* and *nfirelist*, all components of the state described below. If the properties where only guaranteed at the beginning and end of an iteration of the main loop then correctness could not be established.

A more formal analysis could use Hoare Triples or derive a program from the specification. The level of detail given here might be insufficient for that task, especially about the operations like "deliver message to component d" which in reality involves several steps and takes about 100 lines of C. The operation relies upon the truth of and maintains several invariants involving variables not documented in this thesis.

It would also be theoretically possible to describe TCP sockets and the main loop in CSP and show that it is a refinement of the CSP description (possibly by showing they are both refinements of a third system). This would be very complex due to the extensive amount of state and complexity of the operating system interface, even if the model of TCP is simplified to a buffer

with finite capacity.

8.1 Input and output parallelism

The algorithm used by the implementation, described in detail below, appears to have district input and output phases. The purpose of this section is to analyse the impact of the buffering provided by TCP sockets has on this apparent distinction. The discussion has been deliberately simplified by ignoring a number of options not used in by the library, for example socket options and the `sendmsg`(2) system call.

Each connected TCP socket is modelled as a receive and transmit buffer on both hosts. If there is space in the reception buffer then data can be received from a remote host, whether or not the application has called `read`(2), or one of the other system calls that read data from a socket, for example `recv`(2), that can be used to read data from a socket. The `read`(2) system call transfers data from the receive buffer into user space and allows further data to be received.

The buffering of data to be transmitted allows data to be sent even if the remote node's receive buffer is full. Unless special measures are taken the `write`(2) system call will return immediately if all the data will fit in the buffer, even if the data can not be transmitted immediately. The amount written is used to determine which messages where sent. The `__mp__component` structures of the appropriate components are updated. If a component's last output has been sent and it has received an input then it is added to the list of processes to be fired in the next compute phase.

When the space is available on the remote node the data will be transmitted, possibly while the sending host is doing something else (especially with higher specification networking hardware, which can do some of the required processing itself).

If the free buffer space is insufficient, or enough data can not be read, then the default is to perform as much as possible and then block until the operation be resumed. The library uses non-blocking I/O which returns an amount read or written even if it is less than the requested amount instead of blocking. This requires the library to maintain enough state to resume the transmission at an appropriate point.

The library always reads all the data available and delivers it, attaching multiple messages, or partial messages, to a single input if required. This ensures that neither the transmission nor the reception buffer remains full. The use of non-blocking I/O avoids the problems associated with suspended `read`(2) or `write`(2) system calls.

Thus there is no reason why data should not be sent during the reading phase or received during the writing phase. The buffering provided and use of non-blocking I/O results in the parallel input and output required by the CSP model. The serialisation of outputs to components on the same node only affects the timing, due to the buffering of messages on component's inputs. As the proof in chapter 5 shows networks of asynchronous components are deadlock free for *any* possible timing.

The `select`(2) system call is used to efficiently wait for data to be readable from inputs and space to be available for sending data via relevant outputs. The `select`(2) system call adjusts its arguments, which specify which file descriptors the caller is interested in, to reflect which file descriptors are readable and writable. An output is relevant if, and only if, there is unsent data destined for a component on the node at the other end of the output.

8.2 Abstract model of the state

If c is a component and o an element of its state then $c.o$ represents element o of component c's state. The data is modelled using abstract data types which have a 1 to 1 correspondence with the concrete implementation of the state.

Each component c has

- inputs $c.i(n)$, each of which is a, possibly empty, list of input messages. $c.i(0)$ is the first input, $c.i(1)$ the second input, etc.
- outputs $c.o(n)$ each of which is a list with length ≤ 1 of output messages. The index n determines the destination component and input index.
- *nwrite*, the number of outputs that have not been completely sent
- *nread*, which counts the number of read inputs if the component is unable to fire. A component with at least one read input will fire in the next computation phase unless it has an unsent output ($c.nwrite > 0$).

Each node also has a single instance of each of the following:

- A function f that maps destination numbers to a component and input index.
- A list of components, *outlist*, with an unsent output.
- A list of components, *firelist*, of components to fire.
- A list of components, *nfirelist*, of components to fire in the next cycle
- A list of at most two components, *msg*.
- A list of at most one component, *active*.

The most important invariants, many of which can be verified by simple induction, are

1. No component appears more than once on *firelist*, *outlist* or *nfirelist*.
2. $outlist \cap firelist = \emptyset$, $outlist \cap nfirelist = \emptyset$ and $firelist \cap nfirelist = \emptyset$, where the intersection of two lists is defined as the set of elements on both lists.
3. For all components $c \notin msg$, $c.nwrite = \#\{n \mid length(c.o(n)) \geq 1\}$.
4. For all components c, $c \in firelist \vee c \in nfirelist \Rightarrow c.nwrite = 0$
5. For all components $c \notin msg$, $c.nread > 0 \Rightarrow c \in firelist \vee c \in nfirelist \vee c \in outlist$.
6. For all components c, $c \in outlist \Rightarrow c.nwrite > 0$.
7. For all components $c \notin active \cup msg$, $c.nread = \#\{n \mid \text{length}(c.i(n)) \geq 1\}$.
8. For all components $c \notin active \cup msg$, $c \in outlist \Leftrightarrow c.nwrite > 0$.

These invariants hold before and after every statements. Much of the main loop relies on one or more of these facts—if they were not true when delivering a message, for example, the assignment on line 59 might cause the invariant to be violated, and thus the invariant could not be relied

upon anywhere. The lists *active* and *msg* are used to exempt components from some of the facts temporarily.

The *active* variable contains the active component, for which the last two facts may be temporarily false. The *msg* variable contains message endpoints, for which some of the facts are also briefly false due to the separation of message delivery and adjustment of the variables. Lines 41 to 55 arrange for, or perform, message delivery and lines 56 to 57 perform the additional processing to take account of queued messages and restore the penultimate condition.

Since *active* and *msg* are only used by the analysis neither is required in an implementation in a conventional programming language.

8.3 The main loop

The algorithm used by the TCP+ transport main loop is shown below. Termination is achieved by a method that is sufficiently violent that there is no need to handle it in the main loop.

1. $msg = \langle\rangle$
2. $active = \langle\rangle$
3. $firelist = \langle c \mid c \in \text{local components} \wedge c.nread > 0 \rangle$
4. $nfirelist = \langle\rangle$
5. $outlist = \langle\rangle$
6. forever
7. if $firelist \neq \langle\rangle$
8. timeout = 0
9. else
10. timeout = ∞
11. wait for an input from any (remote) node, a writable relevant node or timeout
12. for each readable TCP connection from a remote node
13. read the data available
14. n = desintation number of input
15. $c = f(n)$
16. $msg = \langle c \rangle$
17. Store data in appropriate input of component c
18. if an input has been read
19. $c.nread = c.nread + 1$
20. if $c.nwrite = 0 \wedge c \notin firelist$

21. $firelist = \langle c \rangle \frown firelist$

22. $msg = \langle\rangle$

23. for each writable TCP connection to a relevant remote node

24. while the connection can accept more data

25. c = first element of *outlist* with an unsent output to relevent node

26. $msg = \langle c \rangle$

27. send as much of the data as possible

28. if the message was sent completely

29. $c.nwrite = c.nwrite - 1$

30. if $c.nwrite = 0$

31. $outlist = outlist \setminus \{c\}$

32. if $c.nread > 0$

33. $firelist = \langle c \rangle \frown firelist$

34. $msg = \langle\rangle$

35. $nfirelist = \langle\rangle$

36. for each component $c \in firelist$

37. $active = \langle c \rangle$

38. $c.nread = 0$

39. $firelist = firelist \setminus \{c\}$

40. invoke component body (using a library core function)

41. for each used output

42. if it is sent to a component on another node

43. if $c \notin outlist$

44. $outlist = outlist \frown \langle c \rangle$

45. else

46. d = destination component

47. $msg = \langle c, d \rangle$

48. deliver message to component d

49. $c.nwrite = c.nwrite - 1$

50. if $c \notin nfirelist$ and $c.nwrite = 0$

51. $nfirelist = \langle c \rangle \frown nfirelist$

52. $d.nread = d.nread + 1$

53. if $d \notin firelist \cup nfirelist \cup outlist$

54. $nfirelist = \langle d \rangle \frown nfirelist$

55. $msg = \langle\rangle$

56. if any queued messages exist and $c.nwrite = 0$

57. $nfirelist = \langle c \rangle \frown nfirelist$

58. $active = \langle\rangle$

59. $firelist, nfirelist = nfirelist, \langle\rangle$

The last line is a single, atomic, operation which changes both *firelist* and *nfirelist* to the value of *nfirelist* immediately before the statement and the empty sequence respectively. The $firelist = firelist \setminus \{c\}$ removes all items of *firelist* equal to c. On line 39 the statement removes the first item of *firelist* and no other items. The $\frown$ operator concatenates two lists.

This algorithm should be scalable to a large number of components per node, because there is no global scanning of the component list involved. Two or more nodes can share a single processor on a unix-like system, because the waiting is implemented using a system call that blocks until one of the specified events occurs.

Scalability to a large number of nodes is less clear, because the entire list of components that need to communicate via TCP is scanned in order to collect the message to send to each node via TCP. If the number of nodes is large then each pass might examine many inactive outputs and outputs for other destinations that are not relevant. It would be more scalable to handle the outputs to use per destination message lists, or to send all messages in a single pass through the list of components with unsent messages.

8.4 *firelist*, *nfirelist* and *outlist* are disjoint

Initially, *firelist* is non-empty on at least one node and both *nfirelist* and *outlist* are empty, and thus initially all three lists are disjoint.

Line 21 preserves the property because $nfirelist = \langle\rangle$ at this point, by induction, and line 20 ensures that $c.nwrite = 0$ and thus $c \notin outlist$.

Line 33 preserves the property because it only adds components, c, to *firelist* which statement 31 removed from *outlist*. Similarly statement 57 is also safe because any component added to *nfirelist* has been removed from *firelist* by line 39.

Line 44 is safe because it the component it adds to *outlist* was removed from *firelist* by line 39. Line 54 preserves the disjointedness because it only adds a component to *nfirelist* if it is not in any of *firelist*, *nfirelist* and *outlist*.

Line 51 preserves the disjointness because invariant 6 implies that $c \notin outlist$ and due to the context $c \notin firelist$. The latter is due to c being the active component, which implies that c has been removed from *firelist* by line 39.

The disjointness is preserved by line 59 because the statement assigns the empty sequence to *nfirelist*, in addition to copying its contents prior to the assignment to *firelist*. In practice it is safe

to just assign *firelist* to *nfirelist* because the value of *nfirelist* is not used again until after line 35, which sets *nfirelist* to the empty sequence.

8.5 For all components $c \notin msg$, $c.nread > 0 \Rightarrow c \in firelist \vee c \in nfirelist \vee c \in outlist$.

Initially some components, possibly all of which are on other nodes, have inputs which are the initial stimuli injected into the system and no components have outputs. Statement 3 inserts these components into *firelist* in any order. During the brief period during message delivery when invariants 3, 5, 7 and 8 would be false the components affected are exempted from them by membership of *msg*. If the main loop used multiple threads these regions would be critical sections where the state might be temporarily inconsistent.

When lines 14 to 21 are executed $active = \langle\rangle$ and thus $c.nwrite > 0 \Leftrightarrow c \in outlist$. Formally by induction, at this point $nfirelist = \langle\rangle$ and thus $c.nread > 0 \wedge c.nwrite = 0 \Rightarrow c \in firelist$. Thus if $c \notin outlist$ in line 21 adds c to *firelist* if line 19 increased $c.nread$.

Any component c removed by line 31 with $c.nread > 0$ is added to *firelist* by line 33. Lines 53 and 54 ensures that any component c that has its *nread* increased by line 52 is on *firelist*, *nfirelist* or *outlist*.

Lines 48 and 49 deliver a message from the active component c and adjust $c.nwrite$ accordingly. If $c.nwrite$ is reduced to 0 and $c.nread > 0$ then lines 50 and 51 add c to *nfirelist*. Disjointness is addressed in section 8.4 above.

If line 56 makes $c.nread > 0$ after component c has fired then either $c.nwrite > 0$ and therefore $c \in outlist$, or c is added to *nfirelist* by line 57.

To summarise, invariant 5 remains true because every action that alters any component's *nread* or *nwrite* is matched by a statement that maintains the truth of invariant 5, by appending the component to *outlist* or prepending the component to *firelist* or *nfirelist* when this is required. The affected components are exempted from some of the requirements during these operations by membership of *msg*.

8.6 Line 58 preserves the invariants

Line 58 sets *active* to the empty sequence, which preserves the invariants only if

- For all components $c.nread = \#\{n \mid \text{length}(c.i(n)) \geq 1\}$.
- For all components $c \in outlist \Leftrightarrow c.nwrite > 0$.

This requires a demonstration that the state has been adjusted to take account of the component body's activity. The library function that invokes a component's body returns with `nwrite` set to the number of outputs in use; thus invariant 5 is preserved. After component c has fired $c.nread = 0$ so invariant 8 is also true.

After component c has fired c cannot be a member of *firelist*, *nfirelist* or *outlist*. Since the component was a member of *firelist* before it fired it cannot be a member of either *nfirelist* or *outlist*. Line 39 removed the component from *firelist*.

If the component body sent a message using `mp_output` then $c.nwrite > 0$ and c is not a member of *outlist*. There are two ways of restoring the invariant: reducing $c.nwrite$ by sending

the messages or adding the component to *c.outlist*. The library adopts the former stratagem for messages to local components and the latter for messages sent to components on other nodes. This maximises the effective network bandwidth by the aggregation of multiple small messages into a single large one.

If c sends a component to a remote node then $c.wnrite > 0$ at line 58 because lines 43 and 44 ensure that $c \in outlist$. This prevents c being added to *nfirelist* by lines 51 and 57. Otherwise either line 51 or 54 adds c to *outlist* if c is now fireable, presumably because it sent a message to itself.

Any components which become fireable as result of the delivery of a local message are added to *nfirelist*, unless they are in *firelist* or *nfirelist* already. Correctly handling a component that sends a message to itself and no remote messages requires adding the source component to the list of fireable components if sending a message reduces its *nwrite* to 0 and its $nread > 0$.

Thus after the loop from line 41 to 54 for all components c, $c \in outlist \Leftrightarrow c.nwrite > 0$. *nread* starts at 0 and is increased by 1 for every message delivered to an unused input. Lines 56 to 57 inclusive move all queued messages forward and adjust nread to take account of the dequeued messages. Thus when line 58 is reached for all components c, $c.nread = \#\{n \mid \text{length}(c.i(n)) \geq 1\}$.

8.7 If a message is sent the destination will fire with the message as an input

That a message sent to a component will cause the component to fire with that message as input, and that it will have no unsent messages when it does so, is a fundamental feature of the programming model and therefore must be verified. This depends on several of the properties above, in particular that a component with one or more readable inputs, i.e. $nread > 0$, is either in *firelist*, *nfirelist* or *outlist*.

This requires showing that a component on any of the lists will become the first member of *firelist* within a finite time. In the firing phase, implemented by lines **36** to 57, no component is added to *firelist* and all components of *firelist* are fired. Thus it suffices to show the existence of a firing phase with the component in *firelist*. If the component is in *firelist* there is nothing to prove and if the component is in *nfirelist* then line 59 transfers it to *firelist*.

A component on *outlist* will be transferred to *firelist* when all its outputs are sent if its $nread > 0$. Proving that this will happen requires showing that the relevant TCP connection will accept enough data to transfer the entire message, however large it is and the component will become the first member of *outlist* within a finite time.

Provided sufficient memory is available then each TCP connection can be modelled as a separate buffered connection[1]. Progress is guaranteed because the remote node will always read the available data, and deliver messages contained in it, when it enters its communication phase. The fact that the `mpkern` library always reads data from a readable TCP connection during the communication phase ensures that the TCP connection's internal buffers will not remain full, possibly preventing further progress.

The fact that components are added to the end of *outlist* ensures that a component on *outlist* will reach the front of *outlist* within a finite time, and therefore its outputs will be sent within a finite time. Adding components to the beginning of *outlist* would make it difficult to make the

[1] If insufficient memory is available for this to apply then it is probable that `mpkern` library will kill the job due to an error it cannot handle or memory allocation failure.

bytes received independent of the amounts that can be written in each cycle and make it hard to show that a component in *outlist* will reach the front in a finite time.

Queued messages are guaranteed to be processed because they contribute to a component's *nread* after the previous element of the queue on the corresponding input has been processed. A component with a queued input is added to *nfirelist* if it has no unsent outputs preventing it from firing. A component with queued inputs that cannot fire due to one or more unsent outputs becomes a member of *outlist* with $nread > 0$, which will fire when the unsent outputs are sent; this must happen within a finite time due to the properties above.

8.8 Summary

A detailed description of the main loop of the TCP+ transport, the most important invariants stated and assertion reasoning used to argue that these invariants are maintained. The same facts would support a more formal approach, which would require further details in some areas. The invariants and a progress property were combined to argue the most important property of the CSP model, that if a message is sent the destination component will fire with it as an input, is also a property of the implementation.

Chapter 9

Performance analysis: The Hartree-Fock problem

9.1 The Hartree-Fock procedure

This chapter reports the performance of a real application using the `mpkern` library. As explained above the library's main loop consists of a computation and communication phase, so bandwidth and latency are not appropriate measures of performance. The time between a request to send something and the actual communication depends on how soon the communication phase is entered. For real applications bandwidth is similarly complicated by the effects of aggregation of a (possibly unpredictable) number of messages, which might have unpredictable sizes.

This section avoids these problems by using a real large scale numerical problem, the closed shell atomic Hartree-Fock problem, which retains most of the features of the molecular Hartree-Fock problem. The molecular Hartree-Fock problem has many "real world" applications and large instances are usually solved on supercomputers.

9.2 Overview

In order to demonstrate the performance and utility of the `mpkern` library, a representative application has been selected from quantum chemistry; the solution of the electronic Hartree-Fock equations. The self-consistent field theory embodied in the Hartree-Fock equations plays an important role in many areas of chemistry, atomic and molecular physics, molecular biology, solid state physics and nuclear physics. It is the purpose of this chapter to demonstrate the performance and features of the `mpkern` library, rather than to review the details of computational quantum chemistry, which may be found elsewhere[19, 22, 38].

A simple example of the Hartree-Fock method is investigated here, which is restricted to the solution of the equations for atomic systems. This choice facilitates the detailed tracing of the flow of data around a program of managable complexity while performing computationally intensive numerical tasks which are representative of those which are found in computer packages which solve the molecular Hartree-Fock equations.

9.2.1 Matrix Hartree-Fock equations

The Hartree-Fock method is an independent-particle approximation which is widely used as a model of the electronic structures of atoms, molecules and solids, or as the basis for more elaborate models which correct the short-comings of the independent-particle model by including the correlation of the electronic motions. The spatial solutions of the Hartree-Fock equations for atoms or molecules are called orbitals, $\{\psi_i(\mathbf{r})\}$, which are known as spin-orbitals if a label denoting the spin degree of freedom is appended to the function. For molecules the solutions are almost always obtained by the Rayleigh-Ritz method, according to which each orbital is expanded in a finite basis set of square-integrable functions, $\{\chi_k(\mathbf{r})\}$, so that

$$\psi_i(\mathbf{r}) = \sum_{k=1}^{N} c_k^i \chi_k(\mathbf{r}), \tag{9.1}$$

where the quantities $\{c_k^i\}$ are expansion coefficients, which are real in the present case. The dimension of the representation is denoted by N.

Roothaan[31, 32] was the first to obtain the matrix form of the Hartree-Fock equations for systems constructed from closed electronic shells, by minimising the total electronic energy of the system with respect to variations in the expansion coefficients. This leads to the solution of a generalized matrix eigenvalue equation of the form

$$\mathbf{Fc} = \mathbf{ESc} \tag{9.2}$$

where $\mathbf{F}$ is the Fock matrix, $\mathbf{S}$ is the overlap (or Gram) matrix, $\mathbf{E}$ is a diagonal matrix containing the orbital eigenvalues, and $\mathbf{c}$ is a matrix whose columns contain the expansion coefficients of the orbitals. It is convenient to define the density matrix, $\mathbf{D}$, by

$$D_{ij} = \sum_{a=1}^{N_{\mathrm{occ}}} c_i^a c_j^a \tag{9.3}$$

where the sum over a includes all spin-orbitals which are regarded as being occupied by an electron. The specification of $\mathbf{D}$ depends on the current values of the expansion coefficients, and so the Hartree-Fock method is an iterative procedure. The procedure is repeated until the change in the expansion coefficients is less than a specified tolerance measured with respect to some appropriately defined criterion.

The elements of the Fock and overlap matrices are defined by

$$F_{ij} = h_{ij} + \sum_{kl}[(ij \mid kl) - (il \mid kj)]D_{kl} \tag{9.4}$$

$$S_{ij} = (i \mid j) \tag{9.5}$$

where

$$(ij \mid kl) = \int \frac{\varrho_{ij}(\mathbf{r}_1)\varrho_{kl}(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|}\, d\mathbf{r}_1\, d\mathbf{r}_2 \tag{9.6}$$

$$(i \mid j) = \int \varrho_{ij}(\mathbf{r})\, d\mathbf{r} \tag{9.7}$$

$$\varrho_{ij}(\mathbf{r}) = \chi_i^*(\mathbf{r})\chi_j(\mathbf{r}). \tag{9.8}$$

The one-electron integrals, h_{ij} , involve only the kinetic energy and the external electrostatic fields of the nuclei, and do not depend on the density matrix. These do not change from iteration to

iteration and may be pretabulated if required. Similarly, the integrals $(i \mid j)$ are just the elements of $\mathbf{S}$ and depend only on the choice of basis set.

The two-electron integral $(ij \mid kl)$ represents the electrostatic Coulomb interaction energy between the charge density $\varrho_{ij}(\mathbf{r}_1)$ of electron 1, and the charge density $\varrho_{kl}(\mathbf{r}_2)$ of electron 2, where the overlap charge densities are determined by the basis functions.

If the number of basis functions is N, then the number of integrals of the form $(ij \mid kl)$ which may be formed from them scales as $O(N^4)$, with a cost per integral which depends on the definitions of the basis functions labelled by the set $\{i, j, k, l\}$. In principle one could calculate these once and store them, but this is not feasible even for moderate values of N, in which case they are recalculated as they are needed in the construction of $\mathbf{F}$. In large-scale molecular calculations the number of integrals actually calculated may be reduced significantly by exploiting the eightfold permutational symmetry of the indices, by the use of point group symmetry, and by the use of integral economization algorithms. These algorithms evaluate integrals only if they make a significant contribution to $\mathbf{F}$, using estimates based on strict upper-bounds to the value of the integral weighted by the elements of $\mathbf{D}$ with which it is multiplied. The implementation here exploits index and point group symmetry, but does not include any integral economization algorithms, and thus all integrals are recalculated when required in each iteration.

9.2.2 Methodology

The computational cost in each of the examples presented in this chapter is controlled by the selection of the system, and the accuracy with which the Hartree-Fock equations were solved. Light elements, such as neon, involve s-type and p-type atomic orbitals for which the computational complexity of the resulting two-electron integrals is low. In comparison, a heavy element, such as mercury, involves orbitals of s-, p-, d- and f-type, involving integrals of far greater complexity. The accuracy with which the equations are solved depends both on the specification of the elements of the basis set, $\{\chi - k\}$ and on the dimension, N.

For purely technical reasons involving the evaluation of two-electron integrals, the almost universal choice of basis function in quantum chemistry is a Gaussian amplitude, which is the choice adopted in this study. Each orbtial is approximated by a finite linear combination of spherical Gaussian-type basis functions

$$M[\mu;\, \mathbf{r}] = \frac{1}{r} N_\mu r^{l_\mu + 1} \exp(-\lambda_\mu r^2)\, Y_{l_\mu}^{m_\mu}(\vartheta, \varphi) \tag{9.9}$$

where $Y_{l_\mu}^{m_\mu}(\vartheta, \varphi)$ is a spherical harmonic function, and l_μ and λ_μ are real parameters.

The cost per integral over Gaussian-type basis sets is almost independent of the choice of the exponent value which defines the elements of the basis set, but depends strongly on the angular quantum number, l. The value of N for each of the s-, p-, d- and f -type functions, N_l, can be varied to control the computational load in the integral evaluation stages of the procedure.

Two versions of the program have been developed in order to investigate the characteristics of the problem. A data parallel version of the program which exhibits simple and predictable timing for the various parts of the Hartree-Fock procedure is compared in this chapter with a task-parallel version in which the timing is considerably more complex.

9.2.3 Module description

The most computationally intensive part of the solution of the Hartree-Fock equations using basis sets involves the evaluation of the two-electron integrals, defined by equation 9.6. If Gaussian basis sets are employed, these integrals may be reduced to intermediate functions involving elementary quantities, which are evaluated in batches which are as large as is practicable. The two most significant operations involve the evaluation of the incomplete beta function, $B(a, b;\ x)$, defined by

$$B(a, b;\ x) = \int_0^x t^{a-1}(1-t)^{b-1}\, dt \tag{9.10}$$

for $0 \leq x \leq 1$, and powers of the composite Gaussian exponent $\lambda_{ij} = \lambda_i + \lambda_j$, where λ_i and λ_j define a pair of Gaussian exponents involved in the construction of ϱ_{ij}, equation 9.8. The incomplete beta functions, referred to as β-functions throughout the text, are evaluated in calls to the routine `incomplete_beta`, while the required powers of λ_{ij} are evaluated in `klinit`, with some minor initialisation of the data performed in module `klset`. These intermediates are combined together in module `nre2c1` to form the two-electron integrals $(ij \mid kl)$.

The elements of the Fock matrix are constructed in module `nrfock` according to the prescription defined by equation 9.4, by multiplying the integrals by the current values of the elements of the density matrix, **D**. All other operations involved in the construction of **F**, and the subsequent diagonalization of the representation, represent an insignificant part of the computational effort and will be ignored in the analysis which follows.

9.3 Parallelisation target

The profile of an efficient serial version of the Hartree-Fock problem, applied to a mercury atom, shows that the bulk of the computing time is spent computing incomplete β functions. The 'self' numbers reflect the time spent in the function, excluding functions called within the function. The total numbers reflect the time spent in a function including any functions invoked within the function. Details can be found in the `gprof` documentation[8].

%	self		self	total	
time	seconds	calls	s/call	s/call	name
73.60	2528.22	8788392	0.00	0.00	incomplete_beta
21.26	730.17	189000	0.00	0.02	klinit
4.67	160.40	189000	0.00	0.00	nre2c1
0.44	15.23	288	0.05	11.93	nrfock
0.01	0.42	288	0.00	0.00	klset

All other functions in the profile combined consume 0.02% of the computing time and thus have no significant effect on the running time of the computation. All the functions listed, except `nrfock`, are called only from `nrfock`. Thus `nrfock`, which computes blocks of the Fock matrix, uses almost all the time and `nrfock` is a sensible target for a coarse grained parallel implementation of the Hartree-Fock procedure. A finer grained parallelisation strategy might parallelise the computation of incomplete β functions.

Two approaches to a coarse grained parallel version of `nrfock` are presented here, task parallelism and data parallelism. Task parallelism breaks the problem into a number of independent computations, which can be performed in parallel—each processor executes a different section of

the code, which implements its portion of the computation. Data parallelism executes the same code, or similar code, on all nodes each of which processes a different fraction of the data.

Neither version parallelises anything other than `nrfock`, because the cost of the rest of the problem, including diagonalisation of matrices of small dimension, is not significant.

9.4 Task parallel `bertha`

The task parallel implementation of `bertha` breaks `nrfock` into pieces that can be computed separately. When two results have to be combined this is carried out by a separate component. The task graph is based on a careful analysis of the data flow in `nrfock`. Each matrix, set of normalisation constants, sets of beta coefficients, etc is computed separately. The complex communication graph makes the guarantees of correctness provided by the `mpkern` particularly valuable.

The task parallel version of `bertha` has two types of node. Node 0 is the master node and does little apart from maintaining queues, and run components which must respond quickly to efficiently exploit the possible parallelism. The fact that the library is single threaded makes this incompatible with any computation which might take a significant amount of time.

All the other nodes are worker nodes which do almost all the processing, with only limited reference to the master node. This design obviously makes it impossible to achieve more than a speed up of $n - 1$ on n nodes.

Figure 9.1 shows a simplified diagram of the components of the task parallel `nrfock` implementation, which shows the structure of the Hartree-Fock procedure. All the components are shown and every component shown is a separate component. Connections missing from the figure send long lived data from entry barrier, if the barrier is passed, to many of the compute components. The shaded nodes are only involved once in each invocation of `nrfock` and the unshaded ones are involved more than once. The remainder of the components are very similar to the data parallel implementation.

Many of the components have a significant amount of persistent state, often long lived data with an attached serial number. Compute jobs can refer to the data they want by serial number and thus avoid the duplicate network traffic involved in sending it with every request. The top of loop component can only handle a single instance of `nrfock`. The entry barrier component queues up Fock matrix requests that cannot be serviced immediately and distributes appropriate portions of instances, with a fresh serial number, to the compute components.

The data fusion algorithm depends on monotonically increasing job serial numbers. A serial number queue maintains the long lived data and a general purpose queue maintains a list of jobs. When the component is fired any job included is added to the work queue. The first element of the work queue is processed, provided the required data is available. If the work queue is non-empty the component sends itself an empty message so it fires again, and can process the remainder of the queued jobs. Serial number queues, with built in handling for arithmetic overflow, and the more general queues used to implement the work queue are features of the `mpkern` library.

The collect coulomb and collect exchange components gather complete sets of β-function values and send them all in a single message to the coulomb integral and exchange integral components respectively. The β-function results have routing information, passed on without processing by the beta worker components, that identify the appropriate place to store the result.

Both the collection coulomb and coulomb integral components, and collect exchange and exchange

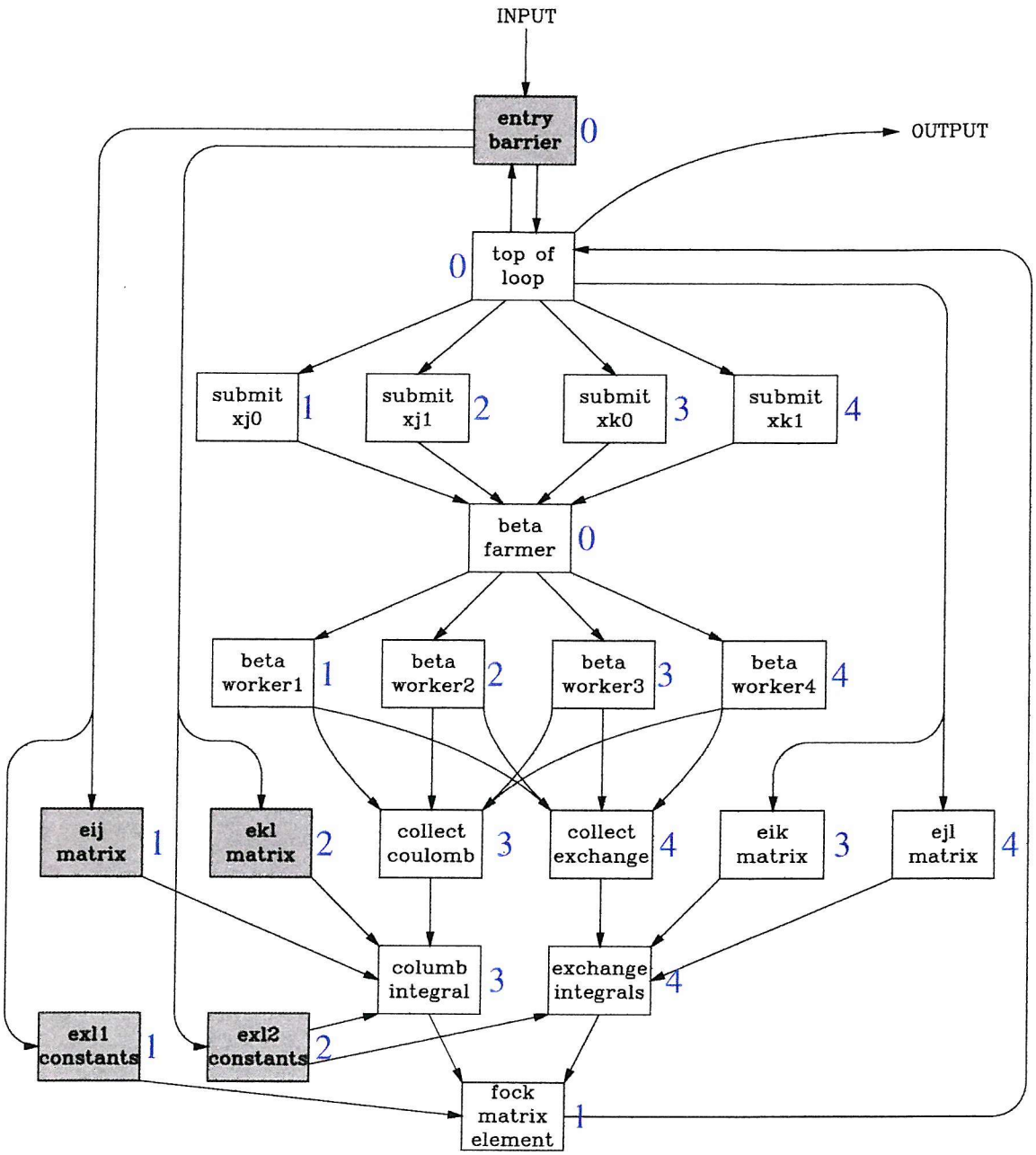


Figure 9.1: Simplified diagram of task parallel `nrfock` (for 5 nodes). Node numbers are shown in blue beside components. Note that this is an **inefficient** distribution on 5 cluster nodes.

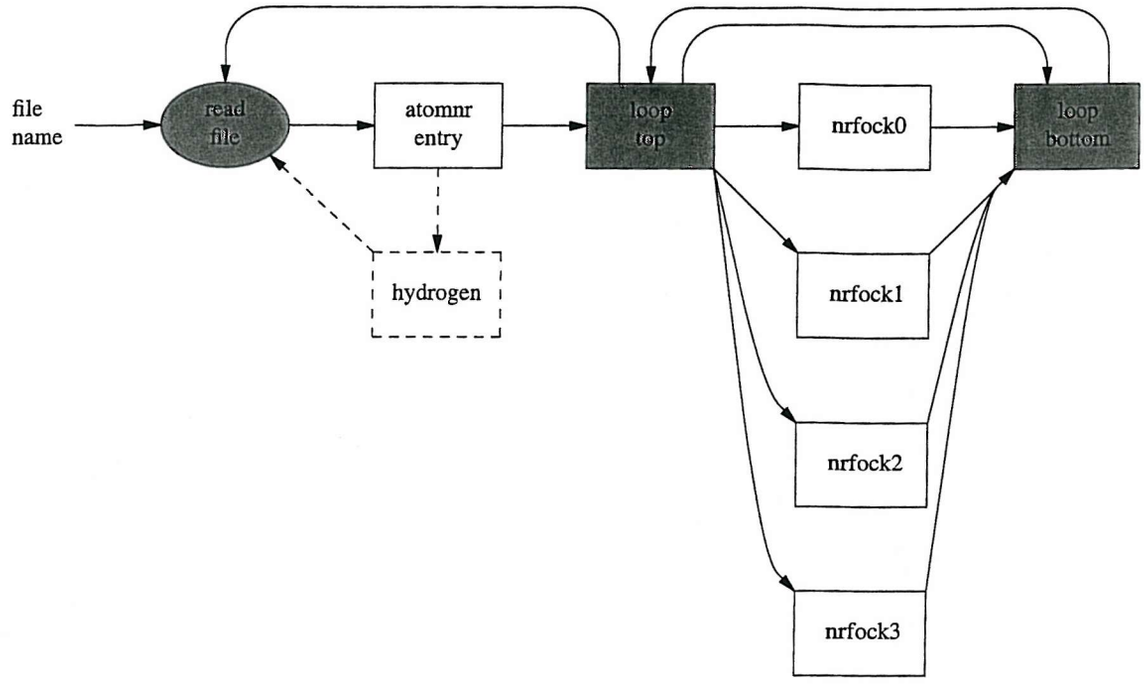


Figure 9.2: The communication graph of 4 node data parallel `bertha`

integral components run on the same node and so the collected results can be transfered using in memory transfer.

9.5 Data parallel `bertha`

Data parallel `bertha` parallelises `nrfock` by dividing the work into portions which can be computed largely independently with little duplication. A small amount of the calculation is duplicated to reduce the complexity of the design and reduce the communication requirements. All the nodes, including node 0, do a significant portion of the task: there is no need for a master node to coordinate the activities of the other nodes. All the components except the nrfock$n$ components run on node 0. The nrfock$n$ components run on node n; for example nrfock2 runs on node 2. The dotted lines show the special case of hydrogen, which does not require any iterations to arrive at the correct solution because only one electron is involved.

The cost of distributing the jobs and collecting the results is not significant, largely because the portion of the computation performed by the nrfock$n$ components takes 99.98% of the time in the serial version. Each nrfock$n$ component is stateless—no information is retained between instances of nrfock$n$. There is none of the complexity involved in tracking serial numbers and the communication requirements are significantly reduced. All the parameters are sent in a composite message, including the area that the component should compute. The computation can take a significant amount of time, especially for larger matrices.

This design requires a well balanced distribution to be computed in advance, because once the nrfock$n$ components have been started the program can only wait for them all to complete. The computational cost of a matrix element is a complex function of many factors.

The work distribution used here can be characterised by $p-1$ boundaries, b_1 to b_{p-1}, where p is the number of nodes. nrfock0 computes elements 0 to $b_1 - 1$, nrfock1 computes elements b_1

to $b_2 - 1$, ..., nrfock$p - 1$ computes elements b_{p-1} to $m^2 - 1$ where m is the size of the matrix being computed (with elements numbered 0 to $m^2 - 1$). There are two methods of selecting the boundaries illustrated here, namely fixed boundaries and movable boundaries which are adjusted to reduce load imbalances.

9.5.1 The fixed strategy

The fixed strategy distributes Nf_i elements to node i, where N is the number of matrix elements, $f_i = t_i^{-1} / \sum_{j=0}^{p-1} t_j^{-1}$ and t_i is the time required to compute a matrix element on node i. Given n identical nodes all the f_i are $1/n$ and the same number of elements is assigned to each node. Allowing boundaries anywhere in a matrix, not just at the start of a row, improves the load balance.

The fixed strategy works well for the tested instances of the Hartree-Fock problem when using identical nodes. The fixed strategy, using speed estimates derived from the wall time[1] taken to compute a standard set of Fock matrix elements, generates the distribution used in iteration 2 of figures 9.4 to 9.8. The slow node, which corresponds to the red markers, clearly finishes ahead of the fast nodes in iteration 2.

A better result can be obtained by adjusting the boundaries so the slow node computes more elements, as shown in the later iterations of figures 9.4 to 9.8, which shows the effect of adjusting the boundaries using the boundary adjustment algorithm below.

9.5.2 Fixed boundary selection

The purpose of this section is to justify computing Nf_i elements on processor i when using p nodes that take times s_0, ..., s_{p-1} to compute N_0 Fock matrix elements respectively. Fock matrix elements are used instead of microbenchmarks, which measure a single aspect of a machine's performance, because of the complexity of computing the cost of Fock matrix elements from microbenchmark results. Only the relative values of s_0, ..., s_{p-1} matter. This analysis also determines an upper bound on the maximum possible speed up using a collection of nodes with known, differing, speeds.

The analysis makes the simplifying assumption that all the elements of a Fock matrix cost exactly the same amount to compute, which is not strictly accurate. All the measured times are scaled, using s_0, ..., s_{p-1} to estimate the time required on node 0. If no estimates are available then s_0, ..., s_{p-1} are all taken to be equal.

Any data parallel solution computes a N element Fock matrix by computing n_0, ..., n_{p-1} elements on nodes 0 to $p - 1$ respectively. The best balanced solution, which might require one or more nodes to compute fractions of an element, satisfies

$$\frac{n_0}{N_0} s_0 = \frac{n_1}{N_0} s_1 = \ldots = \frac{n_{p-1}}{N_0} s_{p-1} \tag{9.11}$$

and $\sum_{j=0}^{p-1} n_j = N$. Together these imply that

$$n_i = \frac{N/s_i}{\sum_{j=0}^{p-1} 1/s_j} \tag{9.12}$$

[1] time as measured by a clock on a wall

and the speed up of the distribution computed using (9.12), assuming the fastest node in the cluster is used to compute the serial time, is

$$S = \min_{0 \le i \le p-1} \left(s_i \sum_{j=0}^{p-1} \frac{1}{s_j} \right) \tag{9.13}$$

9.5.3 The adjustable strategy

The adjustable boundary strategy estimates the time required to compute the ith element, c_i, using wall time differences measured on the node that computes the element in question, and computes new boundaries assuming the estimates are the true cost in the next iteration. If the ith elements costs exactly c_i in the next iteration then the new boundaries are optimal. It might be theoretically possible to do better by using non-contiguous ranges, which would take longer to compute. In practice, after a few iterations the boundaries fluctuate around a stable distribution and well balanced distribution.

The costs of the elements are estimated from measurements of the wall time differences within the nrfock$n$ components. If the nodes are not identical then the times measured on node i are scaled by s_0/s_i, where the s_n is the wall time, as measured by the difference between two calls to `gettimeofday`(2), to compute a fixed set of matrix elements on node n. The measurement of the s_n times can be conveniently carried out in the first iteration, where there is no need to compute a submatrix of the Fock matrix.

After the end of each iteration a new set of boundaries is computed using element costs estimated from the measurements in that iteration. The boundary adjustment algorithm presented here is based on $t_{n\,p}$ which is the time, in μs, required to compute elements 0 to $n-1$ using nodes 0 to p. The time taken for a standardised set of Fock matrix elements on node i is s_i, for $0 \le i \le p-1$. If all nodes are supposedly identical then the benchmark is not performed and all s_i values are assumed to be equal.

The time to compute no elements on nodes 0 to p, $t_{0\,p}$, is obviously 0. The time required to compute elements 0 to n on node 0, $t_{n\,0}$, is $\sum_{i=0}^{n} c_i$. For all $n \ge 1$ and $p \ge 1$

$$t_{n\,p} = \min_{0 \le j \le m} \left(\max(t_{j\,p-1}, (t_{n\,0} - t_{j\,0}) \times s_p/s_0) \right) \tag{9.14}$$

where s_n is the time required to compute an element on node n.

An optimal set of boundaries can be computed from $t_{r^2\,n-1}$, where n is the number of nodes and r the number of rows of the submatrix of the Fock matrix. The boundary b_{p-1} should be the value of j that minimises $t_{r^2\,n-1}$. b_{p-2} should be computed in the manner applied to computing elements 0 to $b_{p-1}-1$ using $n-1$ nodes and so on until all the boundaries have been determined.

The remainder of this section discusses efficiently computing the values of $t_{n\,p}$. For all $0 \le a \le m^2$, $t_{a\,0}$ is easily computed given the costs, c_i. Given $t_{a\,0}$ and $t_{a\,n-1}$, for all $0 \le a \le m^2$ then all the $t_{k\,n}$ values can be computed by computing $\max(t_{j\,p-1}, (t_{k\,0} - t_{j\,0}) \times s_p/s_0)$ for all values of j from 0 to k, and selecting the best. This alogrithm takes $O(m^4 p)$ time and $O(m^2 p)$ space for p processors and an $m \times m$ portion of the Fock matrix.

The time can be reduced by analysis of $t_{r\,p}$. The first lemma shows an obvious, but important, fact that is used in the second lemma to show that

$$f_{n\,p}(d) = \max(t_{d\,p-1}, (t_{n\,0} - t_{d\,0}) \times s_p/s_0)$$

decreases monotonically to a minimum and then increases monotonically. Exploiting this fact

reduces finding the best split point to $O(\log m)$ and thus reduces the time complexity of computing an optimal set of boundaries to $O(m^2 \log(m)p)$.

Lemma 8 ($t_{n\,p}$ is increasing)
$t_{0\,p}$, $t_{p\,1}$, $\ldots$ $t_{m^2\,p}$ is increasing for all p.

Proof.

Define $c_i > 0$ as the cost of computing the ith element of the $m \times m$ matrix on any node.

Base Case

If $p = 0$ then there is only one node, so $t_{n\,0} = \sum_{i=0}^{n-1} c_i$. Thus $t_{0\,0}$, $t_{1\,0}$, $\ldots$, $t_{m^2\,0}$ is a monotone increasing sequence.

Induction step

Assume by induction the result holds for $t_{n\,p'}$ for all $n \leq m^2$ and $p' < p$. Clearly it suffices to show that $t_{n\,p} \leq t_{n+1\,p}$ for all $n < m^2$.

Let S be an optimal way of computing the first $n+1$ elements of the $m \times m$ Fock matrix on nodes 0 to p, which takes $t_{n+1\,p}$ time. The design above shows that S computes elements 0 to D on nodes 0 to $p-1$ and elements D to $n+1$ on node p. Either $D < n$ or $D = n$.

If $D < n$ then computing elements 0 to D on nodes 0 to $p-1$ and $n - D - 1$ elements on node n takes at most $t_{n+1\,p}$ and thus $t_{n\,p} \leq t_{n+1\,p}$.

Otherwise $D = n$ and $t_{n+1\,p} = t_{n+1\,p-1}$. Computing no elements on node p and the first n elements on nodes 0 to $p-1$ takes $t_{n\,p-1} \leq t_{n+1\,p-1}$ by the induction hypothesis.

Thus $t_{n\,p} \leq t_{n+1\,p}$ for all $n < m^2$. The result follows by induction. □

Lemma 8 can be used to prove the times as function of the split point is decreasing and then increasing.

Lemma 9 ($t_{n\,p}$ is v-shaped)
There for all n and p there exists $0 \leq D \leq n$ such that $f_{n\,p}(0)$, $f_{n\,p}(1)$, $\ldots$, $f_{n\,p}(D)$ is monotone decreasing and $f_{n\,p}(D+1)$, $\ldots$, $f_{n\,p}(n)$ is monotone increasing. where

$$f_{n\,p}(d) = \max(t_{d\,p-1}, (t_{n\,0} - t_{d\,0}) \times s_p/s_0)$$

Proof.

Lemma 8 implies that $(t_{n\,0} - t_{0\,0}) \times s_p/s_0$, $(t_{n\,0} - t_{1\,0}) \times s_p/s_0$, $\ldots$ is monotone decreasing and $t_{0\,p-1}$, $t_{1\,p-1}$, $\ldots$ is monotone increasing.

$f_{n\,p}(0) = \max(0, t_{n\,0}) = t_{n\,0}$ and thus there is exists some D such that for all $d \leq D$ $(t_{n\,0} - t_{d\,0}) \times s_p/s_0 \geq t_{d\,p-1}$. If $d \leq D$ then $f_{n\,p}(d) = (t_{n\,0} - t_{d\,0}) \times s_p/s_0$ and therefore $f_{n\,p}(0)$, $f_{n\,p}(1)$, $\ldots$, $f_{n\,p}(D)$ is monotone decreasing.

For all $d > D$ $t_{d\,p-1} \geq t_{D\,p-1}$ and $(t_{n\,0} - t_{d\,0}) \times s_p/s_0 < (t_{n\,0} - t_{D\,0}) \times s_p/s_0$. Thus if $d > D$ then $f_{n\,p}(d) = t_{d\,p-1}$ and thus $f_{n\,p}(D+1)$, $\ldots$, $f_{n\,p}(n)$ is monotone increasing. □

Lemma 9 shows that the following algorithm finds an optimal boundary

```
min=0
max=M
while min < max
    probe=⌊(min + max)/2⌋
    If f_{n p}(probe), ..., f_{n p}(max) is increasing then
```

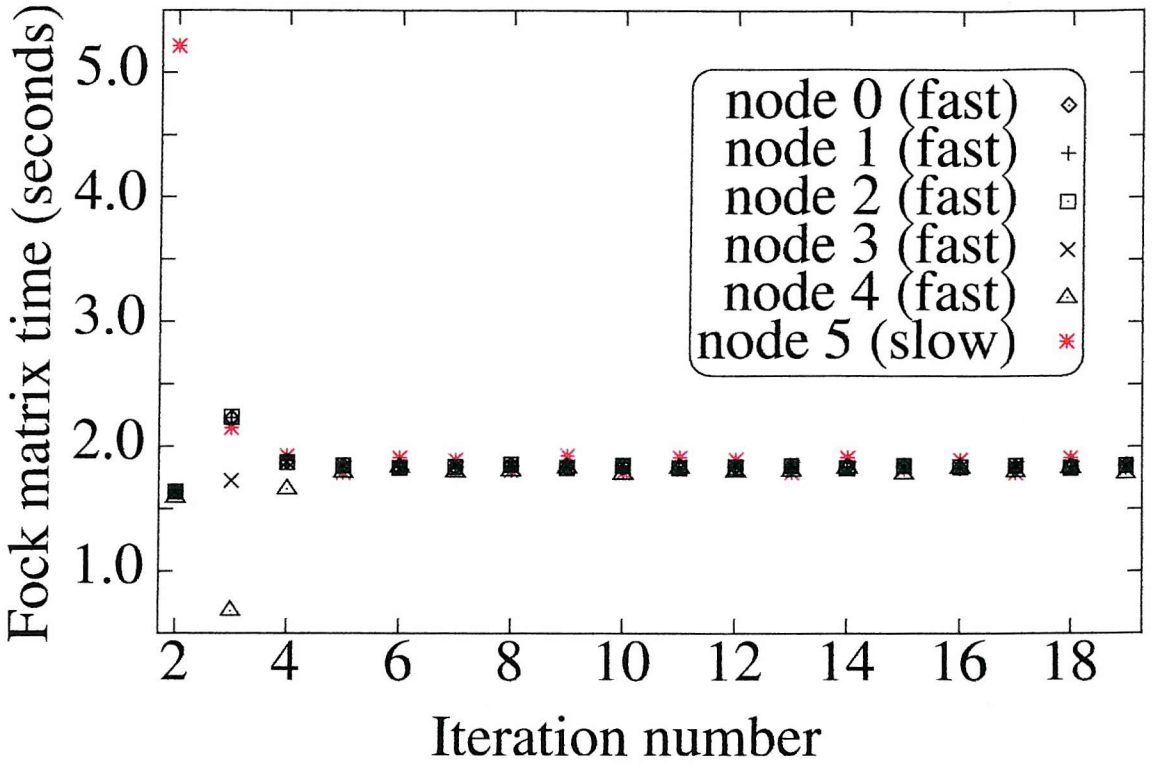


Figure 9.3: Time to compute the first Fock matrix in the "light" instance of mercury with one slow node, with the code assuming equal speed nodes.

```
        max=probe
    else
        min=probe+1
```

where $f_{n\,p}(d) = \max(t_{d\,p-1}, (t_{n\,0} - t_{d\,0}) \times s_p / s_0)$. This alogrithm works by determining which slope of the v-shaped function $f_{n\,p}(d)$ `probe` lies on and adjusting `min` and `max` accordingly.

The algorithm terminates because `max` $-$ `min` is integral, strictly decreasing and bounded below by 0. The loop body's guard implies that `max` $\geq$ `min` $+ 1$ and thus `min` $\leq$ `probe` $\leq$ `max` $- 1$. Therefore both assignments decrease `max` $-$ `min`.

The estimates are based on experimental results, namely two calls to `gettimeofday`(2), and thus are subject to a small amount of noise, due to random factors, for example the time required to transmit a network packet and the amount of CPU time used by background jobs on the node. Fortunately noise is not a critical problem. Rebalancing can produce a good result because if a node is slower than its estimated speed then the costs for its portion are inflated and thus the size of its portion is reduced. The time estimates are only based on the previous iteration.

This decision leads to noisier split points but limits the impact of inaccurate adjustments to take account of the speed differences between different nodes. The conversion is accomplished by multiplying by a constant, based on a single speed measurement, which might be subject to noise and could be too simple to be realistic. The benchmark used is the Hartree-Fock density matrix element computations. The boundary selection algorithm is designed to counteract the deficiencies of the simple cost model.

An illustration of the robustness of the algorithm can be seen in figure 9.3. Figure 9.3 shows

the time to compute the first Fock matrix, for the "light" instance of a mercury atom[2], without taking account of the node speeds, using 6 nodes with node 5 about 1/4 the speed of the other nodes. This inflates the cost estimates of the elements computed on node 5 by about a factor of 4. The unrebalanced results above suggest that all elements have similar computational cost. When some of those elements computed on the slow node are transfered to another node as a result of their high cost elements, their cost estimate falls by about a factor of 4. The balance in iteration 5 and above in figure 9.3 is very good, despite it being based on inaccurate cost estimates.

9.6 Results

Two sets of data were used to assess both parallel versions of the Hartree-Fock problem. A "heavy" instance and a "light" instance of a model of a mercury atom. The "light" instance uses atomic basis sets of size 32, 26, 22 and 21. The "heavy" instance uses larger sets of atomic basis functions, size 40, 38, 29 and 25, and therefore requires more computation. The only difference between the two instances is the amount of computation required (and possibly the accuracy of the computed solution). Henceforth, the less computationally intensive version of the model is referred to as the "light" instance, and the more computationally intensive model is referred to as the "heavy" instance. The program is called `bertha` for historical reasons.

All the timing data is based on wall time as reported by `gettimeofday(2)`. The timer resolution is nominally 1 microsecond but might in reality be coarser. Since both data and task parallelism introduce processing that is not required for a serial implementation, the 1 node case is taken to be the performance of an efficient serial version. The serial version was a clean C version based on an original FORTRAN version, which had a lot of globals, supplied by Dr. H.M.Quiney (who also supplied the input data). The serial version takes 1309.2s (21 minutes 49.2 seconds) for the light instance of a mercury atom and 3940.3s (65 minutes 40.3 seconds) for the heavy instance. These are the numbers to which the parallel versions are compared.

The tabulated results show the time, speed up and per node idle percentages. The "% time idle" column, which is computed as (`wall time` − `cpu time`)/`wall time`, shows the fraction of the time each node is idle, with node 0 at the top left, node 1 at the top right, node 2 at the left of the second line, etc. Summary statistics show the minimum, maximum and average of these figures.

A solution to the same problem based on `MPI` would be sufficiently different to make the results of very limited value for comparison purposes. The purpose of this chapter is not to compare `mpkern` with other libraries, just to assess the efficiency of the `mpkern` library.

9.6.1 Hardware

The hardware used for testing the implementation was a cluster of 6 900MHz athlons, all with a large amount of memory and local hard disc, connected by a dedicated switched 100Mb/s ethernet network. All the systems used the linux operating system. During the testing a slow node, about 1/4 of the speed of the other nodes, was added and this was used for testing the adaptive load balancing algorithm used in the heterogeneous system data parallel results.

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	21 min 43.2s (1303.2s)*	1.00	0.0	0.0	0.0	0.0%
2	54 min 21.6s (3261.6s)	0.40	1.9	48.5	95.0	95.0% 1.9%
3	49 min 47.3s (2987.3s)	0.44	2.6	39.5	92.0	92.0% 23.8% 2.6%
4	47 min 12.0s (2832.0s)	0.46	4.1	49.3	90.4	90.4% 67.2% 35.7% 4.1%
5	44 min 27.2s (2667.2s)	0.49	11.2	57.9	88.3	88.3% 69.5% 73.6% 46.9% 11.2%
6	60 min 38.5s (3638.5s)	0.36	23.9	70.1	88.4	88.4% 57.1% 23.9% 81.6% 85.7% 84.0%

*time taken by efficient serial version

Table 9.1: Task parallel `bertha` results for the "light" instance of a mercury atom

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	65 min 40.3s (3940.3s)*	1.00	0.0	0.0	0.0	0.0%
2	165 min 35.3s (9935.3s)	0.40	1.3	48.5	95.8	95.8% 1.3%
3	152 min 30.2s (9150.2s)	0.43	3.1	42.3	93.6	93.6% 30.3% 3.1%
4	141 min 52.2s (8512.2s)	0.46	3.7	49.7	90.7	90.7% 68.3% 36.0% 3.7%
5	146 min 53.7s (8813.7s)	0.45	9.9	58.6	89.1	89.1% 71.2% 74.3% 48.3% 9.9%
6	176 min 36.6s (10596.6s)	0.37	22.5	70.1	89.1	89.1% 56.4% 22.5% 82.3% 85.8% 84.3%

*time taken by efficient serial version

Table 9.2: Task parallel `bertha` results for the "heavy" instance of a mercury atom

9.6.2 Task parallel results

The performance of the task parallel implementation of **bertha**, as shown in table 9.2 and 9.1, is disappointing, taking significantly longer than the serial time. In practice, the concurrency control requirements and cost of frequent communication between the master and compute nodes are too large and expensive to efficiently exploit the available parallelism. The problem is not much less severe for larger Fock matricies, despite their higher computation to communication ratio.

Methods of attacking this problem, all of which increase the computation to communication ratio, include

- combining several components into a single component, which would reduce the communication and synchronisation requirements (and increase the computation to communication ratio).
- redistributing, and possibly redesigning, the components to reduce the communication requirements between components on different nodes should improve performance, provided the opportunities for parallelism were not reduced too much.
- the complex logic used to handle computing several elements simultaneously in the present implementation is not very scalable. Redesigning it to cost less and scale better should improve the performance.
- exploiting the fact that the same data, thinly disguised by different serial numbers, is sent to the compute components many times. If instead, components stored this data or sufficiently few results computed from it[3], then the communication requirements would be reduced.

Combining all the components into a single component that computes a portion of the Fock matrix, and dividing the Fock matrix between such jobs, results in a data parallel program. However taking some of these strategies to their logical limit need not result in a data parallel program. Instead it might result in a program that can attack portions of several Fock matrices in parallel, which would not be possible in a data parallel program.

9.6.3 Data parallel results

Identical speeds

The data parallel approach works much better when split points are not tied to row boundaries, especially when using a moderate number of nodes. The results when processor boundaries are tied to row boundaries are shown in table 9.3. The improved results in table 9.4 were obtained by allowing processor boundaries at any element.

The 1 node line shows the statistics of the serial version which is CPU bound. In table 9.3 an m row Fock matrix is computed on p nodes by making the first m mod p nodes compute $\lfloor m/p \rfloor + 1$ rows and the remaining nodes compute $\lfloor m/p \rfloor$ rows each[4]. As shown in table 9.3 the idle time steadily increases as the number of nodes increases. Nodes 0 and 1 are always the most heavily used nodes. In no case does any node except nodes 0 and 1 have less than 10% idle time.

As shown in table 9.4 the performance can be improved by allowing the node boundaries to appear at places other than row boundaries. This reduces the differences in the number of

[2] A "light" and a more computationally intensive "heavy" instance were used as benchmarks.

[3] This would exclude, for example, the storage of all m^4 integrals used to compute an element.

[4] If m is a multiple of p all nodes compute the same number of rows

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	21 min 43.2s (1303.2s)*	1.00	0.0	0.0	0.0	0.0%
2	10 min 7.4s (607.4s)	2.16	0.8	1.5	2.1	0.8% 2.1%
3	7 min 13.3s (433.3s)	3.02	1.8	5.5	11.2	1.8% 3.4% 11.2%
4	5 min 29.3s (329.3s)	3.98	0.9	7.6	12.8	0.9% 4.7% 12.1% 12.8%
5	4 min 41.4s (281.4s)	4.65	0.6	13.2	18.9	0.6% 9.5% 18.6% 18.9% 18.4%
6	4 min 15.0s (255.0s)	5.13	9.3	19.5	28.3	9.4% 9.3% 18.4% 23.8% 28.3% 27.9%

*time taken by efficient serial version

Table 9.3: Data parallel `bertha` performance for "light" instance of a mercury atom, with processor boundaries restricted to row boundaries

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	21 min 43.2s (1303.2s)*	1.00	0.0	0.0	0.0	0.0%
2	10 min 9.7s (609.7s)	2.14	0.7	0.7	0.7	0.7% 0.7%
3	6 min 49.2s (409.2s)	3.18	0.8	1.2	1.9	0.9% 1.9% 0.8%
4	5 min 17.6s (317.6s)	4.10	1.4	3.4	5.2	1.4% 3.9% 3.1% 5.2%
5	4 min 8.0s (248.0s)	5.25	1.5	2.1	3.1	2.0% 2.2% 1.7% 3.1% 1.5%

*time taken by efficient serial version

Table 9.4: Data parallel `bertha` performance for "light" instance of a mercury atom, with processor boundaries at arbitrary elements

elements computed by each node to a single element, at most, rather than a whole row. All results below are obtained with code that allows split points at any element without reference to the row boundaries.

After this change, in no case does any node have more than 5.2% idle time and the idle percentages do not necessarily increase as the number of nodes increases. The improvement in the results suggests that forcing split points to be at row boundaries is too inflexible for effective load balancing with more than a small number of nodes.

Similar performance is also observed with a larger problem, the "heavy" instance of a mercury atom, which is used to generate table 9.5, which takes one node a little over an hour.

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	65 min 40.3s (3940.3s)*	1.00	0.0	0.0	0.0	0.0%
2	30 min 55.9s (1855.9s)	2.12	1.4	1.5	1.7	1.7% 1.4%
3	20 min 44.4s (1244.4s)	3.17	1.4	2.1	3.0	1.9% 1.4% 3.0%
4	15 min 37.2s (937.2s)	4.20	1.7	2.3	2.8	2.3% 1.7% 2.5% 2.8%
5	12 min 27.5s (747.5s)	5.27	1.8	2.3	3.1	2.4% 1.8% 2.1% 3.1% 2.0%

*time taken by efficient serial version

Table 9.5: Data parallel `bertha` performance for "heavy" instance of a mercury atom

The times taken for the "light" and "heavy" instances of a mercury atom on identical nodes with rebalancing enabled are shown in table 9.6 and 9.7. The cost of individual elements is estimated by timing each row of the Fock matrix and dividing by the number of elements of the row that is computed by the associated node. These measurements indicate that the cost of the rebalancing for larger cases which are already well balanced outweighs the small benefit that is possible by adjusting the split points.

Mixed speeds

Performing a Hartree-Fock benchmark in the first iteration, namely computing the first N_0 elements of the matrix on each node is worthwhile where the speed of the processors varies widely because the timing allows a much more efficient distribution in iteration 2. Further benefits accrue from the code responsible for the load balancing having a more realistic cost estimate for each element—if a node is a lot slower then it will not inflate cost estimates because the benchmark will determine an appropriate correction.

The distribution computed from the speeds is used in the second iteration in figures 9.4 to 9.8 using equation (9.12). The data in the figures indicates this gives too many elements to fast nodes and too few elements to slow ones. The boundaries computed by rebalancing produce a more evenly balanced distribution. The results in tables 9.8 and 9.9 show an indication of the observed overall performance. The theory numbers are computed using measured speeds and equation (9.13). As a result the estimated theoretical maxima vary between runs of the same problem

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	21 min 43.2s (1303.2s)*	1.00	0.0	0.0	0.0	0.0%
2	10 min 29.7s (629.7s)	2.07	0.3	0.3	0.4	0.4% 0.3%
3	7 min 3.0s (423.0s)	3.08	0.7	1.0	1.2	1.2% 1.1% 0.7%
4	5 min 16.4s (316.4s)	4.12	0.8	1.2	1.4	0.8% 1.2% 1.4% 1.3%
5	4 min 16.3s (256.3s)	5.08	1.0	1.7	1.9	1.0% 1.8% 1.9% 1.8% 1.9%
6	3 min 34.9s (214.9s)	6.06	1.5	2.5	3.0	1.5% 3.0% 2.9% 2.8% 2.4% 2.6%

*time taken by efficient serial version.

Table 9.6: Data parallel `bertha` performance for "light" instance of a mercury atom with rebalancing on identical nodes

Nodes	Wall time	Speed Up	% time idle			Idle %
			min	avg	max	
1	65 min 40.3s (3940.3s)*	1.00	0.0	0.0	0.0	0.0%
2	31 min 47.8s (1907.8s)	2.07	1.4	1.4	1.4	1.4% 1.4%
3	21 min 25.0s (1285.0s)	3.07	1.5	1.7	1.8	1.5% 1.8% 1.7%
4	15 min 59.5s (959.5s)	4.11	1.7	1.8	2.0	1.7% 2.0% 1.8% 1.9%
5	12 min 56.9s (776.9s)	5.07	2.1	2.3	2.5	2.1% 2.4% 2.1% 2.5% 2.4%
6	10 min 48.1s (648.1s)	6.08	2.1	2.6	2.8	2.1% 2.8% 2.5% 2.8% 2.7% 2.7%

*time taken by efficient serial version.

Table 9.7: Data parallel `bertha` performance for "heavy" instance of a mercury atom with rebalancing on identical nodes

using the same hardware.

9.6.4 Summary of results

The task parallel implementation of the Hartree-Fock procedure, which has complex timing, was too fine grained to effectively exploit the cluster. The guarantees provided by using the library are very valuable for such a complex non-acyclic network of components.

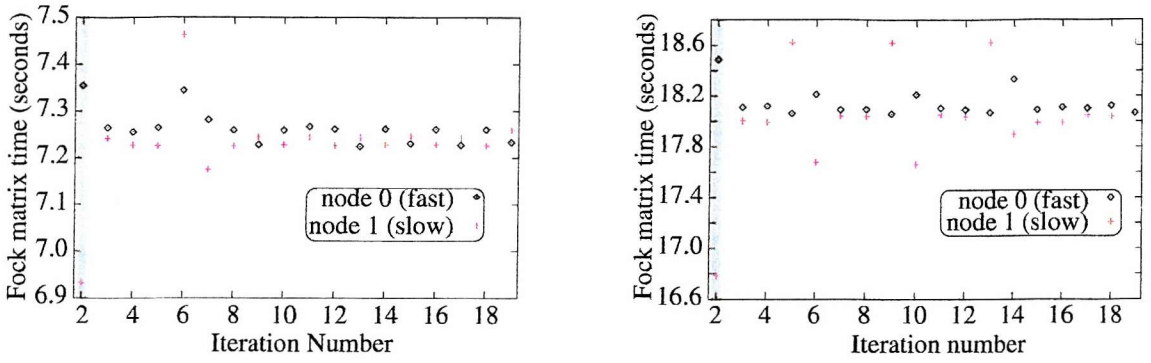


Figure 9.4: First Fock matrix generation times for both "light" (left) and "heavy" (right) instances of a mercury atom for one fast and one slow node. Iteration 2 is highlighted by a blue background.

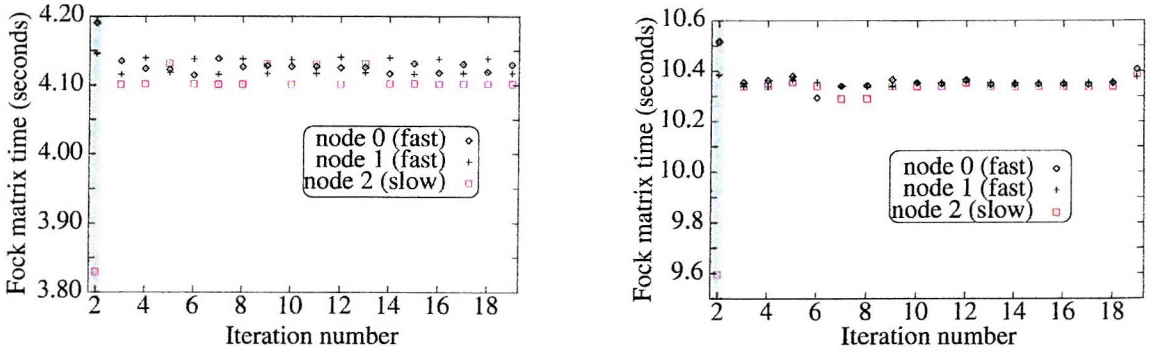


Figure 9.5: First Fock matrix generation times for both "light" (left) and "heavy" (right) instances of a mercury atom for two fast and one slow node. Iteration 2 is highlighted by a blue background.

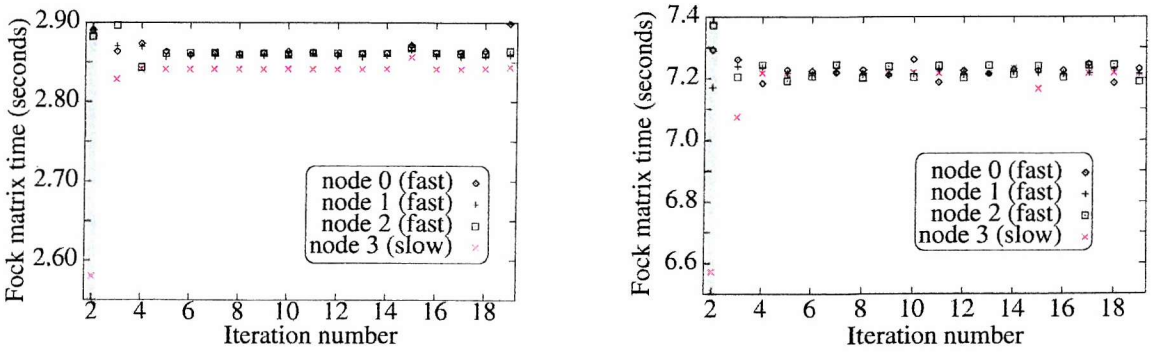


Figure 9.6: First Fock matrix generation times for both "light" (left) and "heavy" (right) instances of a mercury atom for three fast and one slow node. Iteration 2 is highlighted by a blue background.

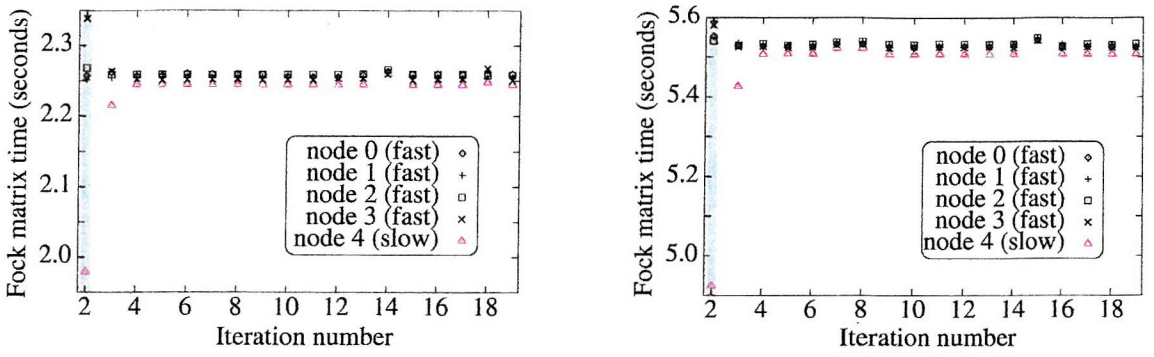


Figure 9.7: First Fock matrix generation times for both "light" (left) and "heavy" (right) instances of a mercury atom for four fast and one slow node. Iteration 2 is highlighted by a blue background.

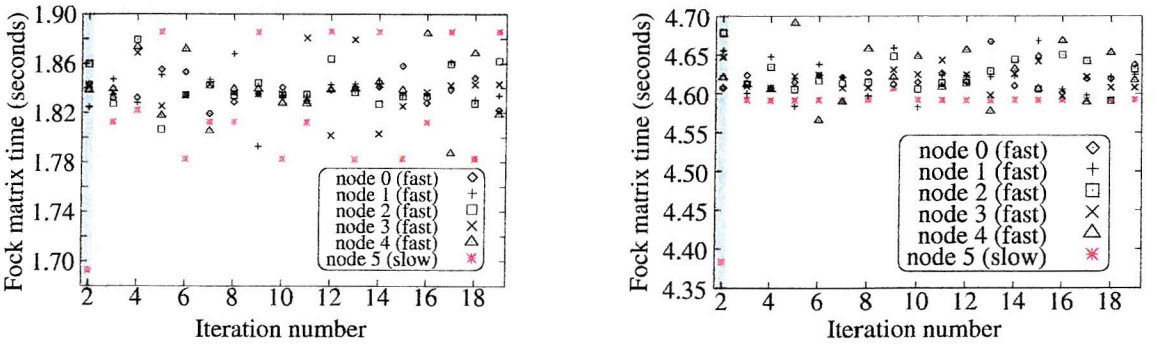


Figure 9.8: First Fock matrix generation times for both "light" (left) and "heavy" (right) instances of a mercury atom for 5 fast and 1 slow node. Iteration 2 is highlighted by a blue background.

Nodes	Wall time	Speed Up		% time idle			Idle %
		Theory	Real	min	avg	max	
1	21 min 43.2s*	1.00	1.00	0.0	0.0	0.0	0.0%
2	16 min 23.0s	1.28	1.33	0.7	0.8	0.8	0.7% 0.8%
3	9 min 17.5s	2.29	2.34	1.7	1.9	2.1	1.7% 2.1%
							1.8%
4	6 min 19.7s	3.27	3.43	1.8	2.1	2.4	1.8% 2.4%
							2.3% 2.0%
5	4 min 53.4s	4.28	4.44	2.0	2.6	3.0	2.0% 3.0%
6	4 min 4.2s	5.23	5.34	3.3	4.1	4.6	3.4% 4.6%
							4.6% 4.3%
							4.6% 3.3%

*time taken by efficient serial version

Table 9.8: Data parallel `bertha` for "light" instance of a mercury atom using 1 slow node

Nodes	Wall time	Speed Up		% time idle			Idle %
		Theory	Real	min	avg	max	
1	65 min 40.3s*	1.00	1.00	0.0	0.0	0.0	0.0%
2	49 min 49.0s	1.27	1.32	0.6	0.6	0.6	0.6% 0.6%
3	28 min 4.1s	2.28	2.34	1.4	1.5	1.6	1.4% 1.6%
							1.4%
4	19 min 0.2s	3.25	3.46	1.3	1.5	1.6	1.3% 1.5%
							1.6% 1.4%
5	14 min 39.6s	4.26	4.48	1.6	2.0	2.2	1.6% 2.1%
							2.2% 2.1%
							1.8%
6	12 min 27.7s	5.25	5.27	2.3	3.3	3.6	2.3% 3.6%
							3.5% 3.6%
							3.6% 3.1%

*time taken by efficient serial version

Table 9.9: Data parallel `bertha` for "heavy" instance of a mercury atom using 1 slow node

The data parallel results show that data parallelism is an efficient way of attacking the Hartree-Fock problem and that the implementation of the `mpkern` library is efficient for a small number of processors.

The results with one slow node indicate that the different performance characteristics of slower processors, even older processors of the same series, can be sufficiently different for simple calculations to produce an inefficient data distribution. The load redistribution algorithm based on lemma 9 on page 89 effectively solves this problem. Use of this algorithm to adjust the data distribution could allow large instances of many data parallel algorithms to make effective use of nodes of a range of different ages and speeds.

Chapter 10

Conclusion and Further Work

Two variations on task parallel programming have been described and proved to be deadlock free. A close relative of the more flexible variation is implemented by the `mpkern` library. The `mpkern` library's efficiency was measured using both a (fine grained) task and a (coarse grained) data parallel implementation of the atomic, closed shell, Hartree-Fock problem.

A technological application of parallel computing which is likely to be of increasing importance is the real-time visualisation of the data generated by simulations, and in the commercial exploitation of high-speed computer graphics. The `mpkern` library might be appropriate for implementing interactive parallel programs if the ability to fire components due to external inputs, for example mouse clicks was added (this would not require any major changes to the present implementation of the library).

The programs modelled are composed of components, which *fire* when input is available. Components only communicate with each other via a fixed set of synchronised, directed connections. A component cannot send a message unless the destination is willing to receive one. Input and output is done in parallel eliminating problems with cycles—all the communications in a cycle succeed, avoiding deadlock. Parallel input and output also prevents the aggregation of components introducing deadlock, as illustrated in figures 2.1 and 2.2 in chapter 2.

Given an arbitrary communication graph, distributed on multiple processors in an arbitrary manner, the need for global analysis is almost completely eliminated by the deadlock freedom guarantees. This simplifies the implementation of programs with a complex communication graph, for example the task parallel Hartree-Fock program. It is sufficient to show that data passes through any concurrency control barriers and that the correct data is processed.

The `mpkern` library is appropriate in situations where the order of events is hard to predict, or event driven parallel programming is appropriate. The library might also be appropriate in cases where correctness is critical, even if the problem is not naturally event driven. In hard real time environments[1], which require responses within a strict deadline, the supplied implementation of the library is not appropriate.

The results of the data parallel implementation of the Hartree-Fock program indicate that the `mpkern` library is efficient. A program using the `mpkern` library efficiently can compete with a similar program written using another library, for example MPI, or in a parallel programming language, for example HPF (**h**igh **p**erformance **f**ortran).

[1] examples of these environments include fly-by-wire controls on aircraft, chemical plant process control and nuclear power station controls

The problem of obtaining a well balanced distribution for mixed speed nodes in the data parallel Hartree-Fock problem was effectively solved by dynamically adjusting the boundaries, in response to the time required in the previous iteration. Since the adjustment algorithm is not specific to the Hartree-Fock problem, it is reasonable to conclude that dynamic load balancing allows data parallel programs to make effective use of heterogeneous resources.

10.1 Evaluation of the library implementation

The `mpkern` library implementation's goals are correctness, efficiency, scalability, flexibility and simplicity. This section attempts to evaluate how well the library core linked with the TCP+ transport meets these goals. The coverage is not exhaustive and mentions a few implementation details that are not mentioned elsewhere in this document.

The performance and applicability of the library for a real parallel program was demonstrated in chapter 9, using a simple version of the Hartree-Fock problem. The task parallel solution performed poorly, probably due to being too fine grained to exploit effectively the (linux) cluster but the data parallel version performed well.

- **Correctness:** The main reasons the library is believed to be correct are
  - The use of non-blocking I/O prevents deadlocks due to blocking during input or output. Thus a node can always enter the computation phase and fire a fireable component.
  - The fact that each component fires at most once during the computation phase ensures that the computation phase of a fireable component will be entered within a finite time, provided that all component bodies are correctly implemented.
  - The buffering provided by TCP implementations and the reading of data from all readable sources ensure that no TCP connection is permanently unwritable (and that messages sent via a TCP connection will be delivered).
  - As explained in the correctness section, an input will always cause a component to fire and progress on sending messages is guaranteed.
- **Efficiency:** Good points of the library when linked with the TCP+ transport, in addition to those listed under scalability, are
  - direct use of relatively simple operating system facilities, avoiding potentially expensive abstraction layers.
  - aggregation of messages makes efficient use of the network and reduces context switching.
  - use of a small number of large transmissions, efficiently exploiting the buffering in most TCP/IP implementations. This allows the `writev`(2) system call to return before the data has been sent (which happens in the background, during the computation phase).
  - batching of many messages into a single transmission allows aggressive minimum delay options to be used (Nagle[25] disabled, minimum delay in the IP type of service field) without generating a large number of small, high overhead, packets.
  - that messages to components on the same node are sent by pointing the destination's data structure at the message, instead of via an interprocess communication mechanism, minimising the cost of these messages.

- not processing the contents of messages, thus avoiding the cost of converting data into an external format and back again, which just wastes processor cycles in a homogeneous environment.
- that a node waiting for the ability to read data from another node, or send it to another, only wakes up due to events that allow progress.
- that no features of the library implementation included to improve scalability obviously reduce performance on small systems.

Bad points of the library, when linked with the TCP+ transport, are

- that the separate communication phase can lead to significant delays between a request to send a message and actual transmission.
- that very large messages, which cannot be completely transmitted in a single communication phase, can cause severe delays to messages sent after them to other components on the same remote node.

- **Scalability:** The design features of the library that enhance scalability include
  - that all the nodes are essentially identical, except during start up and shut down. Thus no node can be a bottleneck in a well distributed program.
  - the examination of only active components after the first list of fireable components has been generated.
  - mapping of a destination number to a component and input index is based on an efficient balanced tree structure.
  - a single threaded design, avoiding context switching when many components run on a node.
  - per destination message counts, which are inexpensive to maintain, make building the list of nodes which need to be contacted very fast.
- **Flexibility:** Linking the core with a transport substantially reduces its flexibility, because it fixes the initial phases, main loop and termination routines, all of which are part of the transport specific code. However some flexibility remains including that
  - any remote shell command and file name of the host list can be used. If the defaults are wrong then setting appropriate environment variables will override them.
  - programs are self contained and can thus process the argument list in any way they see fit, provided they are robust enough to cope with the arguments added by the library on the remote nodes, for example by calling `mp_init`, before processing them.
  - the library only reserves one argument, which entirely consists of non-ASCII characters, for its own use[2]. Programs are free to interpret any other arguments in any way they see fit.
  - a program can specify the maximum number of nodes to use or, albeit with difficulty, construct a list of components and connections between them based on the number of nodes supplied. Programs designed for a fixed number of nodes always work on fewer nodes, albeit not necessarily efficiently.

[2]Even this argument will be passed to the program unless it is in the position the library expects to find it.

- the existing support for multiple protocols, and both deferred and immediate transport mechanisms, in the TCP+ code should make it relatively simple to add support for other protocols.

- **Simplicity:** The library has some subtle points but it could be much more complex. The design decisions that simplify the library include:
  - A single threaded implementation avoids the concurrency control that would be required in a multiply threaded implementation.
  - The implementation uses relatively simple and low level communication facilities. This avoids the complexities that would arise if a complex, and probably expensive, abstraction layer was used.
  - All the data structures are relatively simple and well understood.
  - The library just passes messages between components. This avoids all the processing required to convert data into a predetermined format before transmission, and reversing the process when a message is received.
  - The adoption of a format independent of how much data can be sent at any time, and the absence of any "negotiation" about the available messages and which of them the receiver has resources in place to handle.

- **Portability:** The library builds with no changes on solaris and linux. This requires the build system to provide `inet_aton` on solaris but not on linux, where it is part of the standard library.
  - The `configure` script, generated using `autoconf`, uses only portable shell features and determines determines the existance of functions, header files and probes for other system and compiler features required.
  - Implementations are provided for most functions which might not be provided, for example `inet_aton` and `alloca`. These functions are used automatically on systems on systems that require them.
  - The library is built using `libtool`, which knows how to build both shared and unshared library on a wide range of unix variants.

10.2 Further work

The Hartree-Fock example only solves simple cases of the Hartree-Fock problem and does nothing with the result. It should, at least in principle, be easy to extended it to an efficient solution of the molecular Hartree-Fock problem and add appropriate processing of the results for a range of applications.

At present the `mpkern` library fails to take advantage of shared memory on multiple processor nodes, which should be a lot faster than the complex TCP protocol, which is designed for a possibly unreliable network. Other missing features include

- special support for fast response components. The current implementation cannot support slow components and components that must respond quickly on a single node. This can make it difficult to achieve high performance implementations of some designs,

for example a process farming implementation using the spare processing power of the farmer's node for computation.

- starvation deadlock detection (no other form of deadlock is possible, by the proof in chapter 5).
- limiting the number of messages sent to a specific input. At present stopping and restarting sending messages to an input is not implemented, making it possible for the messages queued on an input to use an unbounded amount of memory.
- the ability to send multiple messages per cycle via a given output. This is equivalent to inserting buffers between two components and thus the analysis above still applies.
- semi-automatic load balancing. The library provides no load balancing—the load is only well balanced if the programmer distributed the components in a way that is well balanced. Semi-automatic load balancing, either static or dynamic, would be a useful extension.
- the ability to fire a component when an external input becomes readable. This could, for example, be used to implement parallel programs with a graphical user interface. If such an input could be reasonably regarded as infinitely readable the deadlock freedom result above would imply *complete* deadlock freedom.
- compact statistics generation. While it is possible to generate statistics from the output generated by a high verbosity level this involves reducing a large amount of information. Given the programming model it should be possible to record the total time spent in each component, and similar overall statistics, in a compact format.

It is also worth noting that lemma 9 in chapter 9 should apply to many data parallel situations. It should therefore be possible to apply the dynamic load balancing algorithm described in chapter 9 to many data parallel problems. A parallelising compiler could apply the algorithm automatically to suitable loops that are not parallelisable, for example control loops in iterative programs. This would have the added benefit of allowing the resulting executable to make efficient use of nodes with a range of speeds and architectures.

10.3 Availability

Stable releases of the `mpkern` library described above, and the latest development version which might include unstable changes, are both available. If a stable release is downloaded please also download and check the separate PGP signatures to ensure your copy does not include unauthorised modifications. All packages include reference documentation, an automatic configuration script for unix-like systems, and the sample sort and Hartree-Fock examples. The `mpkern` library's home on the world wide web is `http://www.sourceforge.net/projects/mpkern`. Programs may be linked to the `mpkern` library without any restrictions; however changes to the `mpkern` library itself must be made freely available or not distributed at all. The programs and `mpkern` library have only been tested using x86 linux systems, but should work in other similar environments.

The author's (1024 bit) PGP public key, available from PGP public key servers, has a fingerprint of `A5 71 00 EA 8D 64 04 69  5B D2 00 24 EB 28 22 3A`. The increasing computing power of single processor systems has resulted in the author creating a longer (1536 bit)

PGP public key with the fingerprint `D2BA 046C 9385 BF2D F0A7  3323 86D6 B50D BB3E F1A1`. Both these keys will normally be used to sign releases of `mpkern`.

Appendix A

cg grammar

This appendix shows a simplified grammar for the cg language. The only change is that the complex precedence manipulaton and grammar rules that implement constant folding during parsing have been eliminated. The grammar presented here will accept an non-constant value where *constant-number* or *constant-string* appears, which will cause a parse error in the current cg implementation.

It would be simpler to implement constant folding in the actions, which are performed when a reduction via a rule is performed by the parser, and later phases where this is not possible. Performing constant folding in the grammar adds significant complexity to the grammar (especially the rules and precedence mapulation used to ensure that `foo-3-5` is implemented as `foo-8`).

Operator precedence is implemented using precedence declarations rather than being built into the grammar. These declarations are not shown in the grammar here. The dangling `else` ambiguity, which `if` the `else` in `if ... then ... if ... then ... else ...` is part of, is resolved by associating the `else` with the closest previous `if`. The format of the grammar here is based on the input format required by `yacc`, a popular LALR (look ahead left to right) parser generator and the C grammar in the back of *The C programming Language*[17].

The words in *italics* are non-terminal symbols. A flush left non-terminal symbol followed by a : introduces the definition of a non-terminal. An indented line is a sequence, or sequences, of terminal and non-terminals that can be reduced to the closest preceding non-terminal introduced. Alternatives are separated by new lines of |. Thus the first four lines of the grammar describe a *program* as a *program* followed by a *function-declaration,* a *program* followed by a *constant-declaration*, or the empty string (ϵ).

The words and symbols in `typewriter` are terminals given literally. The words in sans-serif are other terminals, for example explicit-string is a terminal which matches both `"I am a string"` and `"I am also a string"`. ϵ represents the empty sequence of tokens.

program:

program function-declaration

program constant-declaration

ϵ

constant-declaration:

`fix nodes`=*constant-number*

`fix num` explicit-string=*consant-number*

`fix string` explicit-string=*constant-string*

function-declaration:

function-type constant-string(*parameters*) *function-body*

function-type:

`void` | `tap` | `num` | `int` | `string`

parameters:

void

parameters, parameter-decl

ϵ

parameter-decl:

parameter-direction var-type explicit-string *parameter-dimensions*

parameter-direction:

direction

parameter-direction direction

direction:

`in` | `out`

var-type:

`string` | `num` | `int` | `tap`

parameter-dimensions:

parameter-dimensions [*constant-number*]

parameter-dimensions [ ]

ϵ

var-dimensions:

var-dimensions[*constant-number*]

ϵ

function-body:

statement

;

statement:

{ *statement-list* }
basic-statement

statement-list:
statement-list statement
statement

basic-statement:
type explicit-string *var-dimensions*;
`if` (*number*) `then` *statenent*
`if` (*number*) `then` *statenent* `else` *statement*
`for` numvar *index=number* `to` *number statement*
`for` numvar *index =number* `to` *number* `by` *number statement*
`while` *(number) statement*
tapvar *index=tap*;
strvar=*string*;
numvar=*number*;
`print` *number*;
`print` *string*;
`print` *tapvar*;
`print` *strvar*;
`print` *numvar*;
void-function*(args)*;
`connect` *endpoint* `to` *endpoint connect-options*;
`component` *string* `body` *constant-string* `on` *number component-options*;

endpoint:
string(number)
tapvar *index*
tap-function*(args) nowhere*
`nowhere`

connect-options:
connect-options `size` *number*
connect-options `type` *string*
connect-options `via` *string*
connect-options `name` *number*
connect-options `constant` *string*
ϵ

component-options:
`with data` *string*
ϵ

tap:

tapvar *index*
`from` *string(number)*
`to` *string(number)*
tap-function*(args)*
`nowhere`

string:
`""`
explicit-string
(*string*)
string-function(*args*)
string`[`*number* `to` *number*`]`
string + *string*
string + *number*
string + *tap*
number + *string*
tap + *string*

number:

comparison

calculation

comparison:

string < *string*

string > *string*

string == *string*

string <= *string*

number < *number*

number > *number*

number == *number*

number >= *number*

number <= *number*

number >= *number*

calculation:

explicit-number

`nodes`

number-function (*args*)

-*number*

+*number*

(*number*)

number || *number*

number && *number*

number | *number*

number & *number*

number + *number*

number - *number*

number % *number*

number / *number*

number * *number*

number ** *number*

args:

arg

args, *arg*

ϵ

arg:

number

numvar *index*

string

strvar *index*

tap
tapvar *index*

index:
index[*number*]
ϵ

constant-number:
number (with no non-constant terms)

constant-string:
string (with no non-constant terms)

The following rules summarise the lexical tokens

- All keywords and operators are lexical tokens.
- Strings and numbers are explicit-string and explicit-number respectively.
- A function declared that returns `void` is a void-function.
- A function declared that returns a number is a number-function.
- A function declared that returns a string is a string-function.
- An integer variable is a numvar.
- A string variable is a strvar.
- A tap variable is a tapvar.
- A declared number constant is an explicit-number.
- A declared string constant is an explicit-string.

The following restrictions apply

- A tapvar is only an endpoint if it is a single tap (not an array of taps)
- A tap must be assigned to either an explicit reference or another tap before its value is used.
- The number and types of arguments must match the parameter, including the sizes and numbers of dimensions of arrays.
- `nowhere` is only allowed as a source.
- The dimension `[]`, which is only allowed in arguments, matches a single argument dimension of any size.
- The left hand side of an assignment must be a single value. Only output parameters can assign whole arrays, or array slices.
- The valid range of an index is 0 to $n-1$, where n with the size of the dimension. Thus if `foo` is declared `num foo[3][5]` then its second index must be from 0 to 4.
- A slice of an $m > n$ dimensional array can be passed to an n dimensional output parameter by specifying the first $m-n$ indices. The dimensions of the array slice are the unspecified dimensions of the array.

- A program must define the function `main`, which must take no arguments and return `void`.
- A function must be declared before it is referenced and all the definitions must match. Using just ; as the function body declares a function exists. A function of the same name with the same arguments with a body must appear later in the `cg` program.

Bibliography

[1] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, 1975.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, Reading, MA, 1988.

[4] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, (17):50–62, 1996.

[5] R. Cleaveland and S. Sims. The NCSU concurrency workbench. *Lecture Notes in Computer Science*, 1102:394–397, August 1996.

[6] Pierre Collette and Cliff B Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. Technical report, Department of Computer Science, University of Manchester, 1995. Technical Report UMCS-95-10-3.

[7] Jim Davies and Steve Schneider. An introduction to timed CSP. Technical report, Oxford, 1989.

[8] Jay Fenlason and Richard Stallman. *GNU gprof: the GNU profiler*, 1992. Inlcuded in the the GNU gprof distribution.

[9] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12), December 1996.

[10] Per Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, August 1993.

[11] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 5(5):425–448, August 1993.

[12] He, Miller, and Chen. Algebraic laws for BSP programming. In *EUROPAR: Parallel Processing, 2nd International EURO-PAR Conference*. LNCS, 1996.

[13] High Performance Fortran Forum. HPF-2 scope of work and motivating applications. Technical Report CRPC-TR 94492, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1994.

[14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 18(2):95, February 1975. Corrigendum.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985. ISBN 0-13-153289-8.

[16] John H. Howard. Proving monitors. *Communications of the ACM*, 19(5):273–279, May 1976.

[17] B. W. Kernighan and D. M. Richie. *The C programming language.* Prentice-Hall, second edition, 1988. ISBN 0-13-110362-8.

[18] M. S. Lam. Current status of the SUIF research project. *Lecture Notes in Computer Science*, 1132:65–78, 1996.

[19] A. R. Leach. *Molecular Modelling: Principles and Applications.* Addison-Wesley-Longman, 1996.

[20] J. Liebeherr and I. F. Akyildiz. Deadlock properties of queueing networks with finite capacities and multiple routing chains. *Queueing System: Theory and Applications*, 20(3–4):409–431, 1995.

[21] W. F. McColl. BSP programming. In G Blelloch, M Chandy, and S Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, May 1994. American Mathematical Society.

[22] R. McWeeny. *Methods of Moecular Quantum Mechanics.* Academic Press, San Diego, 1992.

[23] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. ISBN 0-13-115007-3.

[24] R. Milner. *Communication and Mobile Systems: the π-calculus.* Cambridge, 1999. ISBN 0521643201 (hardback) and 0521658691 (paperback).

[25] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[26] S. Ockiwi and D. Gries. An axoiomatic proof tehcnique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[27] P. Paczkowsi. Ignoring non-essential interleavings in assertional reasoning on concurrent programs. In *Mathematical Frontiers of Computer Science*, number 711 in Lecture notes in computer science, pages 595–607. Springer, 1993.

[28] J. Postel. Transmission control protocol. STD 7 (or RFC 793), September 1981.

[29] J. Quemada, A. Aczorra, and D. Frutos. A timed calculus for LOTOS. In *Proc. 2nd Int. Conf. on Formal Description Techniques (FORTE'89).* North-Holland, 1989.

[30] J. S. Reeve and M. C. Rogers. MP: an application specific concurrent language. *Concurrency: Practice anmd Experience*, 8(4):313–333, May 1996.

[31] C. C. J. Roothaan. New developments in molecular orbital theory. *Reviews of modern physics*, 23:69, 1951.

[32] C. C. J. Rootmhan. Self-consistent flied theory for open shells of electronic systems. *Reviews of modern physics*, 32(2):179–185, April 1960.

[33] A. W. Roscoe. *The theory and practice of concurrency.* Prentice-Hall international series in computer science. Prentice Hall, London, 1998. ISBN 0136744095.

[34] A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. Technical Report Technical Monograph PRG-57, Oxford University Computing Laboratory Programming Research Group, 1986.