Wisdom: A Prototype Scalable Operating System
(Extended Abstract)

Paul Austin, Kevin A. Murray, Andy Wellings
Department of Computer Science, University of York
York, England, UK, YO1 5DD
e-mail: murray@minster.york.ac.uk

Abstract
This extended abstract gives a brief overview of a scalable parallel operating system called Wisdom. This paper outlines the objectives and goals of Wisdom: to provide a usable scalable system, and explains how the components of Wisdom were chosen and designed. The results so far obtained from the prototype are presented and explained, and then analysed to see how near their aims the components come. The experiences of the chosen design methods, architecture and implementation problems are then described, and finally conclusions are drawn.

I. Introduction
Wisdom is an experimental operating system designed to make the use of arrays of processors straightforward and efficient for providing a general purpose computing environment. The objective of the research was to decide upon a scalable architecture and design an operating system that could efficiently use the architecture chosen, regardless of the number of processors present, and correctly reallocate resources (processing, communication and others) as the size of the network changed (either up or down). The idea was that a computer could be constructed from simple building blocks (typically comprising of a processor, memory and some communication links), that could be combined to form a computer of the power needed for a given site or application. Should that site or application then need to increase the power, either to support more 'users' or to allow applications to execute more swiftly, then all that would be required is to add more building blocks. The operating system would then make use of these additional components by placing parts of programs on these new components, or using them to support new users. The requirement that the architecture be easily scalable, together with the need for good interconnection characteristics, made an cartesian mesh (currently two dimensional) the optimal choice as this structure allows a simple routing algorithm and avoids the complexities of many other networks.

Given the objectives of Wisdom, it must, therefore, have the following characteristics:

- **Usability:** The system should provide an environment in which the user can program without needing to supply details about the hardware (which may change), and the environment should be left unaltered by changes in the (size of the) underlying hardware.
- **Flexibility:** In a general purpose computing environment it is not possible to predict all the resource needs of programs and programmers accurately, therefore Wisdom must be flexible enough to support a range of applications and resource requirements.
- **Scalability:** As one of the goals is to use a scalable architecture, the operating system itself must be scalable -- that is it must make use of any new processors without increasing the load on the original processors, and, as far as possible, divide the load evenly between all the processors.
- **Exploit Parallelism:** As a result of scalability, the system must be parallel itself and encourage the use of parallelism. Otherwise, scaling the system up by adding more processors has no advantage since, unless the additional processors can perform some of the work, it will not improve performance.

II. Design
A major element in the philosophy of Wisdom is that of minimalism, i.e. the system comprises of a very small number of components which themselves provide only the minimum support necessary to make the mesh, or processor array, usable and scalable. The choice of the foundation components for Wisdom was taken from a study of many distributed operating systems, including V3, Amoeba1, MOSIX2 and Mach10 and from the work of the ANSA1 project. Part of the ANSA project defined eight forms of transparency that are necessary to conceal the consequences of distribution from the components of applications and programs, and so allow the system to be viewed as a whole rather than a collection of independent systems; these forms of transparency are:

- **Access Transparency:** communications with local and remote objects appear the same
- **Location Transparency:** the location of objects are hidden
- **Concurrency Transparency:** objects may be used, by many other objects, without these other objects being aware of each others' use of the initial object
- **Replication Transparency:** multiple instances of an object may exist with out this replication being apparent
- **Failure Transparency:** object failure is hidden from other objects
- **Migration Transparency:** objects may move within a system without affecting the system

---

TH0337-6/90/0000/0106$01.00 © 1990 IEEE
Performance Transparency: objects are maintained in such a way as to optimise the performance of the system
Scaling Transparency: the system can be resized without its components needing to be altered

An early decision was that the prototype would not deal with the areas of reliability and availability: if the prototype showed that the initial ideas and concepts were feasible then these areas would be considered in future research. Of the above, replication and failure transparency deal with reliability and availability, and hence they are ignored in the prototype version of Wisdom. Concurrency transparency is a property of an object, not the system, as it is the object itself that implements this transparency. This form may, or may not, be desirable, depending on the resources being managed, e.g. file accesses may wish this, but access to the raw disk may not. Migration, performance, and scaling transparency can all be supported, at least partially, with the same mechanism: load balancing. When correctly implemented load balancing allows objects to migrate transparently, increases the performance of the system, and allows the system to be easily scaled. Access transparency is an aspect of routing, or the underlying IPC mechanism: it is this that hides or reveals the remote or local nature of other objects with which the communication is taking place. Location transparency can be provided by naming, since a name should, ideally, give no hint as to the location of an object it names; the system converts the name into a location. Thus the support a system gives, ignoring the issues of reliability and availability, can be described by the support of three areas: parallelism transparency — as many processors as needed exist, interconnection transparency — all processors are fully interconnected, and location transparency — any processor can locate any other willing processor. Each of these elements is supported by one component, or module, of Wisdom: parallelism by load balancing, interconnection by routing and location by naming.

A complete description of all the design issues of the three main components is given in Murray's thesis, but a brief overview will be given here of the major issues in the design of the router and load balancer. The objective of the router is to provide a communication mechanism between the many tasks that may be running. It was decided that routing tables should not be used — they have many difficulties associated with them, primarily maintaining, updating and generating them, as well as potentially becoming unacceptably large for a scalable system. The method of avoiding routing tables is done by labelling all processors with their (x,y) coordinates, and the router function compares the destination with the current (x,y) pair, and can hence work out the appropriate direction to send the message. The environment envisaged had tasks communicating over channels, and these channels would often be passed between tasks. A consequence of this is that channel transfer ought to be a cheap operation. This affected the choice between a virtual circuit and datagram service: virtual circuits are more complex and costly to set up and destroy whereas a datagram needs only destination information — hence the router provides a datagram service. There are two main methods by which data can be transferred: store and forward or wormhole. The wormhole routing method is far faster than that of store and forward; this speed in obtained by performing many small communications (in parallel) rather than a single one. However, if initiating a communication has a significant overhead (as is the case with software) then the wormhole routing method becomes less attractive, since wormhole routing involves many more small communications (albeit in parallel) and this would consequently place a far higher load on the interconnecting processors than store and forward. For this reason store and forward was used in preference to wormhole routing (though hardware should use the wormhole, or similar, routing method). It was also decided that deadlock avoidance was important, and a proof has been published which shows that the router cannot deadlock, a version of which is present in Murray's thesis.

As with routing there are many alternative ways of performing load balancing, each of which have their own advantages and disadvantages. Fortunately there are many excellent surveys of the area which make the advantages and disadvantages of the various methods clearer. Wang and Morris have provided a very good and detailed comparison of the many ways in which a suitable processor for a task may be decided upon, and conclude that the simpler methods provide good results with low overheads and that more complex methods which should provide more accurate results, give only small improvements at higher costs. Eager et al. compare the two main methods of load balancing: sender-initiated in which the processor with the task starts the balancing process and receiver-initiated in which processors without work to do initiate the search for work. Based on this, other literature, and the results of some load balancing simulations a sender-initiated load balancing mechanism was chosen. The destination processor for a task was selected from the current processor and its four neighbours (on a cartesian mesh) based on a comparison of the number of active tasks on the current node and those of its neighbours. A 'watermark' was introduced, which requires that the difference in the number of tasks be greater than a certain value before the balancing takes place, so that the total number of tasks that were transferred and, hence, the likelihood of conspired thrashing, or 'woggling', could be reduced.

III. Performance

The objective of the load balancer is to minimise the imbalance in load over the network, that is it tries to ensure that all the processors have the same number of tasks to execute. Ensuring that the processors have an equal share of tasks helps to give all tasks an equal share of the processing power. This can be considered as minimising the load imbalance, which is defined as:

$$\sum \left[ \frac{L(p_j) - \frac{J}{N}}{2} \right]$$

where \( L(p_i) \) = no. of tasks at \( p_i \)

\( J = \) total number of tasks in the system

\( N = \) total number of processors in the system

When tasks are balanced the distance they are from other tasks with which they wish to communicate will have a direct effect on the performance of the system due to the effect this distance has on communication speed and network load. This is minimized
by balancing tasks only to the immediate neighbours of the processor that spawned them, since it is assumed that tasks are most likely to communicate with their parents and siblings.

![Diagram of processor network](image)

Figure 1: Processor Network used to generate figures 3 to 5

- ▲ 1 processor in test
- ■ 2 processors in test
- ● 3 processors in test
- × 4 processors in test
- ♦ 5 processors in test

Other nodes in network

Figure 2: Legend for figures 3 to 5

Presented here are the results of some simple programs aimed at testing the characteristics and behaviour of the load balancing mechanism. The various programs were run on a network varying in size from one to five processors. The programs run had different communication and computation requirements, and results were collected for each class of programs with the variables being the number of the programs that were executing in parallel, and the number of processors available to receive them. A diagram of the network used, with the processors eligible for receiving tasks is shown in figure 1; figure 2 gives the legend for the succeeding three graphs (figures 3 to 5). When communications take place, they are between a pair of sibling tasks; every task was given a 'twin' for this purpose when created, and the twin was balanced independently. The results shown are for computation-bound tasks (figure 3), tasks which spent half their time in computation and half their time communicating, (figure 4) and finally communication-bound tasks (figure 5). Although not shown in the graphs, the experimental results also indicated that the cost of creating a task remotely was the same as creating a local task. The reason was that the processing costs were shared between the two processors, and that the hardware allowed the communication to proceed whilst the necessary processing was taking place.
compute intensive programs (figure 3), which is as expected. It is also apparent that where there are a number of communicating tasks, the dynamic balancing will also improve their behaviour. This is caused by the computational overhead of communications, which the load balancer shares out, thereby improving the behaviour of communication intensive tasks executing on overloaded processors. A test program was also run on the entire network of nine processors which would perform one of the following operations with the following probabilities: dying - 10%, computing - 50%, communicating - 20%, replicating - 50% (up to a maximum of 50 tasks in the system). As the task ran the likelihood of dying was slowly increase at the expense of the other operations until it reached 50%. During this time the load imbalance, as defined earlier, of the network was calculated at regular intervals (approximately ½ second), and the average imbalance (as defined above) over 100 of these intervals was 1.84.

![Figure 6: Task and Network Configuration for graphs 7 and 8](image)

The objective of the router was to provide high-speed communication between tasks in Wisdom in such a way as to ensure that there was no excessive load placed on the nodes through which messages were travelling and that non-overlapping communications - i.e. communications which do not include any common links - did not interfere with one another. The latter is the most important characteristic since it is in this that communication on a mesh differs from that provided by a broadcast media such as an ethernet. To show that these characteristics were met, and that the router handled high loads well (i.e. it did not suffer from a performance drop under heavy loads) various tests were run which are presented below. These tests involved a number of explicitly placed tasks which communicated 256 byte messages in pairs; figure 6 shows their placement for the graphs in figures 7 and 8. Figure 7 shows the effects of the distance a message has to travel on the time taken to deliver it (with the differing lines in the graph), as well as how increasing network load on the paths that it may take alters the delivery time. Although all tasks were permanently communicating, if there was only a single pair of tasks there is a certain amount of idle time during which the messages were in transit. Increasing the number of tasks causes the processors to be continually creating and receiving messages, and under these circumstances the results show that the router behaves well and is capable of routing messages over other unloaded paths, where
synchronising the start times of the tasks involved in the test. In this figure the graphs shown with the dashed line are equivalent to those in figure 7, and those with the dotted line represent a similar configuration, but with a duplicate set of tasks also placed in the three-by-three mesh in such a way as not to overlap. Figure 9 shows the behaviour of the communication where the routes of the messages do overlap, with figure 9 giving the layout of the tasks for this test. The dashed lines in this graph are a replication of those given in earlier graphs to allow a comparison of the delivery times for overlapping and non-overlapping communication. These results show several facts: firstly if the communications overlap, but there are alternative paths that do not, then the messages will be delivered in such a way that the effects of the overlapping communication are virtually insignificant. Secondly that there is a saturation point reached for message delivery per processor, which with the prototype router is about 1000 messages per second. The results also show a slightly higher cost of receiving messages over simply routing them through the processor; this is due to the coding of the router, and the limited amount of work a task receiving a message performs.

The results presented were taken from the prototype router before any work was done on providing high speeds – this work is currently underway and initial hints seem to suggest that times will be halved. Despite that the results for the communications are not unacceptably slow for a small network. The importance of the results, however, is not their absolute speed, but rather the relative speeds and the effects of scaling that matter. The importance of these results is that they show the absence of interference with messages travelling over non-intersecting paths.

The above results show that the router and load balancer have the required behaviours and characteristics for the limited size prototype that is currently being developed. Evaluating these components to decide on the scalability of Wisdom is more difficult. It is obvious that, since the load balancer only places tasks on its immediate neighbours, a program which creates all its component tasks on a single processor will not directly benefit from an increase in size above five processors. This limit can be overcome either by programming task creation in such a way that tasks are not always created on the same node, or by increasing the neighbourhood to which tasks are balanced, i.e. look at neighbours both directly connected and those that are a single hop away. However, both of these methods would need faster communications to make them worthwhile. The speed of the router, especially if the improved version gives the indicated improvements, would seem to make a network of a few hundred processors (i.e. message travelling a maximum of a few tens of hops) should be possible. The speed improvements should also make load balancing neighbourhoods of 13 processors (i.e. immediate neighbours and one hop away) reasonable. Given this it would seem acceptable to expect that systems of a few hundred processors could be built and used. It should be noted, however, that the limit to the size is a combination of factors. Firstly the speed of communication which routing hardware would improve. Secondly, the number of parallel components which exist in the system (which is largely controlled by the programmers, though also related to the number of users). Finally is the ability to
place the parallel components on free processors, and hence make use of the processors in the system, which is dependent on the load balancing neighbourhood and on the method by which programs create tasks.

IV. Experiences
The experiences, or lessons learnt so far, divide into three basic categories, together with areas that have been shown to need further work. The three main categories are those of the hardware, software and, finally, design and implementation methods.

In the arena of hardware, implementing Wisdom has shown that there are several facilities that would improve the suitability of hardware for this style of system: communications coprocessor, memory protection and mapping which may be accessed by the communications coprocessor, ability to manipulate process queues, conventional cache(s) and fail-handle operation modes. The last three represent the experience of using the INMOS transputer which provides hardware process support, late process queues, conventional cache(s) and fail-handle operation modes. The communication coprocessor is necessary since the speed at which communications can be supported by software is significantly lower than can be supported by hardware, and the speed of communication is very important to the ability to built large networks. In fact, communication chips are already available, though they are not yet coprocessors, and the idea of a coprocessor to speed communication can be justly compared to the idea of a floating point coprocessor to speed floating point operations.

The prototype was implemented in occam⁶ which was developed from CSP⁵ for the transputer, and aimed at providing a provable language for embedded systems. As a result it was a static language which caused severe problems for implementing an operating system, and the language had to be modified somewhat (primarily the ability to send channels - communication points - over channels). But despite these problems the ability to analyse the occam programs⁹ allowed working code to be synthesised to determine if a more efficient equivalent was possible. Occam is not, really, a language well suited to implementing operating systems, but the ability to analyse code is very useful. The area of languages for parallel systems, including the implementation of operating systems, is one that still needs much investigation.

The design, that of selecting the few simple components that are essential to the system, and implementing them as independent modules has proved to be very beneficial. It has allowed individual components to be changed or improved without affecting the remainder of the programs, or needing to alter them. The individual components have remained independent and simple (none of the modules has exceeded 1500 lines of occam code), which has made their implementation and maintenance easier than would otherwise have been the case. The three basic components have served well as a foundation for the system: the programs used for the load balancing tests were the same for all the tests, only the load balancer was altered to inform certain processors not to accept tasks. The test programs then used the router and names to deliver messages irrespective of the destination, and to find the terminal line on which the results where displayed.

Research is currently in progress which aims to provide a parallel file system. This is based around the concept of file agents which provide local caching of file data, and therefore reduce the message traffic and load on the physical file store(s). Provisional simulations have given in very favourable results.

V. Conclusions
The results from the prototype show that it should be possible to build a system which can be scaled from a single processor to a few hundreds of processors, and perhaps beyond, if enough hardware support can be given. This means that the goals of Wisdom can be met, at least up to a reasonable size, representing perhaps a few thousand MIPS of usable processing power. The design method has also proved very flexible: modifying any component has proved easy, due to the simplicity and small size of the code involved, and due to the lack of interference with other parts of the system. The minimalistic approach has succeed in making implementation a relatively simple matter – it took one person about six months to implement and fully debug the three basic modules – and in keeping the size of the system to a minimum, thereby freeing resources for the programs that run under it.

VI. References
