University of Leeds

SCHOOL OF COMPUTER STUDIES

RESEARCH REPORT SERIES

Report 97.1

Algorithm Design and Analysis Using the WPRAM Model [1]

by

Jonathan M. Nash, Martin E. Dyer & Peter M. Dew

January 1997

[1] To be published in High-Level Parallel Programming Models and Supportive Environments (HIPS'97)

Abstract

The takeup of parallel computing has been hampered by the lack of portable software. The BSP model allows the design of portable code for regular computations. This paper describes the use of the WPRAM model to support more irregular problems. A shared queue data type is described, which provides predictable and scalable performance characteristics. The queue can be used to structure the sharing of data in a parallel system, resulting in code which is portable and amenable to performance analysis.

1 Introduction

In the past, achieving high performance on parallel machines required the programmer to exploit the detailed architectural features of the particular platform. The resulting prevalence of non-portable code has severely limited the growth of the parallel software industry. There has recently been a convergence in the features of a parallel architecture, due to the availability of powerful and cheap commodity processors, and the use of scalable high performance networks. A requirement for the development of portable software is the ability to characterise a machine in terms of its global communications capabilities, rather than the specifics of the network interconnection topology. This implies the characterisation of a parallel machine using a *flat* communications mechanism, in which all processors are equidistant. Platforms which support these characteristics include the Cray T3D/E and IBM SP2/3.

There is now a requirement for a standard computational model which can characterise the essential features of these machines, to allow the design and analysis of portable and scalable algorithms. The Bulk Synchronous Parallelism (BSP) model [12, 4] is one such example. The main quality of the BSP is its simplicity, both in terms of the conceptual model and of the associated costing method. A computation is characterised by a sequence of *supersteps* in which communication and computation is carried out independently. Each superstep can be costed using the performance characteristics which measure the network granularity g and barrier synchronisation time L. The cost of an algorithm is simply the sum of its superstep costs. The solution of a problem using supersteps allows the resulting algorithm to be simply costed (see the next section), and has advantages for debugging and performance monitoring.

There are problems for which the BSP seems less appropriate. Algorithms which make use of pointer referenced shared data structures, or where the synchronisation patterns between processors can vary dynamically, are not naturally suited to the use of supersteps. The WPRAM model (*Weakly coherent PRAM*) [10] aims to provide a small set of realistically costed operations which can efficiently support these problems, as well as the important class of bulk synchronous computations. A wide range of problem classes have already been studied [6, 1, 10]. A potential disadvantage is the increased complexity of the cost analysis, since synchronisation in arbitrary groups, the structure of which may rely on runtime dependencies, is an intractable problem. The problem can be tackled by structuring the code using well defined abstractions, to allow costing to be undertaken in a stepwise fashion. Figure 1(a) demonstrates the framework used for the WPRAM. Shared abstract data types (SADTs) [3] provide a mechanism for sharing information in a parallel system, and have a well defined interface and semantics. These allow many of the WPRAM's extended features to be hidden, resulting in user code which essentially operates in a superstep fashion.

This paper concentrates on an analysis of a queue SADT [9]. The queue supports predictable performance to the software layers above, with the performance of its operations being independent of the access of the queue by other processors. Section 2 describes the WPRAM support for bulk synchronous parallelism, together with the associated cost model. Section 3 presents the extended WPRAM operations. Section 4 describes the use of these operations to support the queue, with predictable and scalable performance. The section presents some performance characteristics for the queue, based on analytical and simulation results. The code fragments throughout this paper are based on the notation used in C.

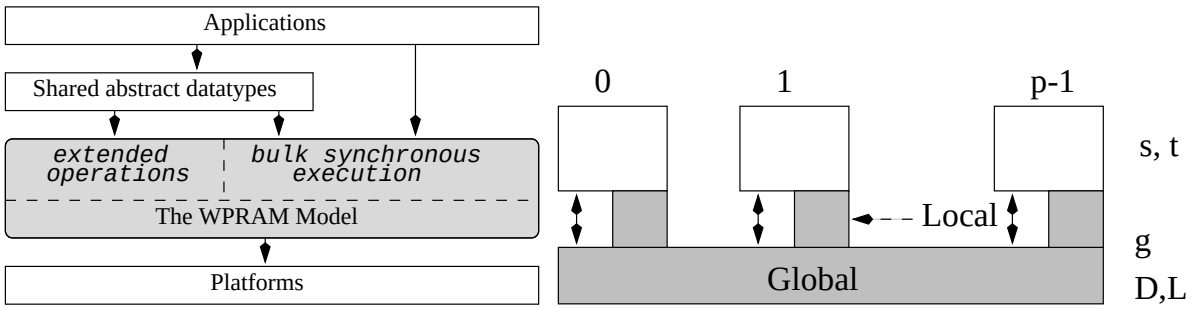


Figure 1: (a) The Approach to Algorithm Design and Analysis; (b) The WPRAM Memory Model

2 WPRAM superstep operation

The WPRAM supports the ability for all processors to barrier synchronise, in the same manner as the BSP model, which divides a computation into a sequence of supersteps. Figure 1(b) shows the communication mechanisms used by the WPRAM, which is based around a shared address space. Global shared data is accessible at a uniform performance by all processors. Local shared data has improved performance when accessed by the processor which allocated the data. The latter can be taken advantage of when data locality is an important issue for the variables being accessed. Global shared memory acts as a repository for data which has dynamic runtime access patterns which cannot be easily predicted. Shared data is also weakly coherent [2], so that it is only guaranteed to be consistent between a set of processors if they explicitly synchronise in some way, such as through the barrier operation.

The WPRAM supports an associated cost model which allows the analysis of algorithms at design time. All costs are normalised with respect to the time, s, of a single local operation (nominally a floating point computation). The other costs are as follows:

- t: The time to transfer a word within the local memory of the processor.
- g: The minimum period between remote memory accesses of successive shared words.
- D: The latency between requesting a shared word and the result returning.
- L: The time taken to execute a barrier operation across all processors.

g and L are defined in an analogous manner to the BSP. L could be defined using g and D to further simplify the cost model. In a given superstep, the operation of processor p_i can be characterised by the following variables:

- w_i: The number of local computations.
- $loc_i[0..p-1]$: Where $loc_i[j]$ is the number of local shared words requested from processor j.
- glo_i: The number of requested global shared words.

The costing of a superstep can then be expressed as:

w+L+gh+tn, where

$w = max_{i=0}^{p-1} \{ w_i \}$

$h = max_{i=0}^{p-1} \{ h_i \}$

$h_i = h_i^{send} + h_i^{recv}$

$h_i^{send} = \frac{p-1}{p} glo_i + \sum_{j \neq i} loc_i[j]$

$h_i^{recv} = \sum_{j \neq i} (\frac{glo_j}{p} + loc_j[i])$

$n = max_{i=0}^{p-1} \{ \frac{glo_i}{p} + loc_i[i] \}$

The term of h_i represents the total number of words leaving (h_i^{send}) and entering (h_i^{recv}) processor p_i. The costing assumes that global shared data is spread evenly across the machine, using some randomising hash function [7]. The value n measures the number of words being transferred within the local memory

of any processor. It can be noted that the computed cost is at most a factor of three greater than the true cost, since the terms w, gh and tn can be equal but due to the execution of three unique processors. In practice this is unlikely, since there is typically either a mixture of all three cost components on each processor, or one component dominates. The cost of a computation is simply the sum of the individual supersteps. A completion time can be derived by multiplying through by a factor of s. This is similar to the costing employed by the BSP model, although it encompasses both the usual *direct* access method and the *automatic* mode of operation [12]. The automatic mode is typically used to emulate the PRAM model [12].

3 Extended features of the WPRAM

The superstep operation described in the previous section forms the basis for expressing parallelism at the coarsest level. This section presents an overview of the support within the WPRAM for more irregular and fine grain forms of parallelism, together with an explanation of how these features fit into the existing cost model.

3.1 Data dependencies

It is an advantage if each processor is able to independently guarantee the completion of a sequence of data accesses. For example, this enables the support of list-based operations, where a processor needs to be able to deference a pointer independently. In the WPRAM model, this is supported using the *switch* operation.

x = y; switch();
A[x] = ... ;

The execution of *switch* guarantees that all previous shared data accesses made by that processor have completed, allowing the results to be used within the following expressions. Within a superstep, if processor p_i executes c_i switch operations, the cost of the superstep can be expressed as:

w+L+gh+tn+Dc, $\quad where\ c = max_{i=0}^{p-1}\ \{\ c_i\ \}$

Again, it can be seen that this is at most a factor of four from the true cost, although this is unlikely, since terms of D imply that a processor is also involved in shared data access, incurring costs related to g.

3.2 Point-to-point synchronisation

List-based operations also require some coordination between the processors. This is most naturally supported using point-to-point synchronisation. The WPRAM supports the idea of *tag* variables, which operate between a pair of processors, also guaranteeing shared data consistency (more formally, release consistency is supported [2]). A tag is initially in an *unset* state, and a processor waiting on such a tag will suspend its execution. Another processor can *set* the tag, which will unblock the waiting processor, if one exists (placing the tag back in its default unset state). The operations which achieve this are $tag\_set(t)$ and $tag\_wait(t)$, for some tag variable t. Complementary operations $tag\_write(t, x)$ and $tag\_read(t, \&x)$ will also transfer a shared address value x as a side effect.

The cost of individual tag operations can be characterised as $D + g$ in the simplest form. This can be added into the expression for the cost of an algorithm. However, superstep operation allows such a simple representation of the cost because the processors are constrained to execute in a stepwise fashion. Within each step it is then known what data access requests each processor is executing. Tags can potentially lose this advantage, since arbitrary groups of processors can now execute independently. In order to preserve the simple costing method, the tags need to be used in a structured manner. Section 4 will describe one such method, based around the construction of a shared queue data structure.

3.3 Concurrent atomic operations

The WPRAM supports a set of operations which allow the concurrent update of a shared variable, without resorting to locking. An example is the addition of an integer value i to a shared integer I. A

processor executing $l = read\&add(I, i)$ gives the following atomic sequence:

$$l = I;\ I = I + i$$

Under the concurrent execution of this operation, results are returned according to some arbitrary ordering. Also, $l = read\&swap(I, a)$ exchanges a shared address a with the current address held in I.

$$l = I;\ I = a$$

The implementation of the operations can either make use of hardware support within the network or software methods to provide scalable performance [7]. It is assumed here that they incur the same costs as for regular data access. These operations allow the construction of shared data structures which can support a high degree of concurrency [9, 5]. The overview of the queue, described in the following section, shows how the operations can support concurrent access to shared array and list structures, and how this can be used to construct the queue data structure.

4 A concurrent queue shared data type

The WPRAM has been used to derive an implementation of a shared queue, which allows all processors to simultaneously enqueue and dequeue items. A detailed implementation of the queue is given in [9]. The semantics of the queue under concurrent access can be defined as follows:

- If processors perform queue operations with no explicit synchronisation between them then the ordering of the operations is as follows:
  - Between processors it is arbitrary.
  - Within a processor it follows the program ordering.
- If processor set P^x synchronises with P^y, such that P^y performs queue accesses after P^x then:
  - If items I^x were inserted by P^x then any queue deletion operations performed by P^y will retrieve items from I^x in preference to items inserted by P^y.

Setting $P^x = P^y = \{p_0..p_{p-1}\}$ supports superstep consistency for queue operations. Setting $P^x = p_i$ and $P^y = p_j$ (for $i \neq j$) supports tag variable consistency.

4.1 An overview of the queue implementation

It is required that the performance of the queue scales linearly with the number of processors p, subject to a degradation incurred by the network latency. More formally

$$f{=}cp/D,$$

for a constant c and frequency f of queue insertion & deletion.

A possibility is to use *read&add* to concurrently access a static shared array, as shown in Figure 2(a).

```
l = read&add (I, 1);
switch();
A[i mod n] = .... ;
```

Each processor retrieves a unique integer value (assuming that the integer has a large enough range of values, so that an overflow will not occur). It can then use this value, modulo the array size, to access an array element. If the number of elements equals the number of processors then each can proceed concurrently. Each array element can then be used support a queue insertion and deletion operation. However, this method cannot be directly used to support the queue abstraction, since the number of outstanding insert or delete requests may exceed the size of the array. So there must be some means of dynamically allocating space at each array element in order to store references to inserted data, and requests for the deletion of data from the queue.

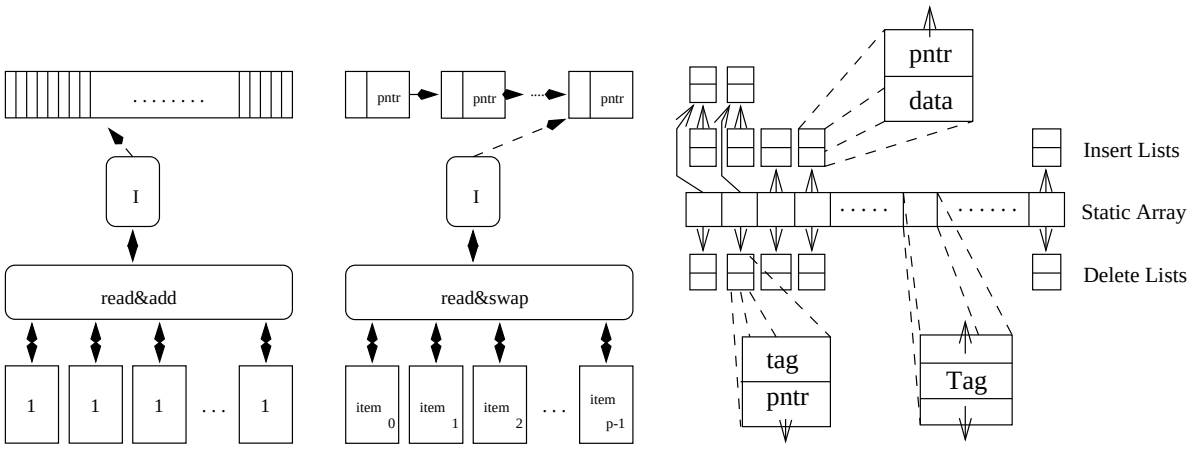


Figure 2: (a) Array and List Access; (b) Queue Data Structure

The *read&swap* operation can be used to support the concurrent insertion of items onto a shared linked list, as shown in Figure 2(a).

```
struct list_item {
        void *pntr;
        .... ;
} item;

l = read&swap (I, &item); switch();
if (l != NULL) l→pntr = &item;
```

Each processor updates the head pointer I by swapping the address of the new item, and is returned the previous item (with the *pntr* field updated if the item exists). However, using this to directly support the queue does not give scalable performance, since list item deletion occurs serially.

A solution is to use both structures - the array to provide concurrent access and the list supporting dynamic storage (Figure 2(b)). Array elements support two lists: one for inserted items and another for pending deletes. A processor accessing the queue will first find an array index using *read&add* (there are two indices - one for each list pair), and then acess the a list using *read&swap*. A tag variable at each array position is used to coordinate interactions between each list pair. A tag within a delete request item is used to suspend the requesting processor until an inserted item becomes available. A detailed description of the queue insertion and deletion procedures is presented in [9].

4.2 An asymptotic analysis of the queue performance

Using irregular forms of parallelism has the potential for an increased complexity of costing analysis. However, a number of queue implementation characteristics allow a simplified costing.

- *Uniform data access patterns*: Storing the array, lists and tags as global shared data results in uniform access times. The queue can store references to local shared data items to support data locality within an algorithm.
- *A high level of concurrency*: There are no implementation aspects which result in serialisation between processors. This is due to the use of concurrent atomic operations to access the static array and the lists.
- *Data-independent access times*: Queue items can be inserted or deleted in a constant number of steps, independent of the number of items already present. This is again using the concurrent atomic operations to access the head of a list, and using tag variables to "pair off" inserted items with pending delete requests.

These features allow the times for queue operations to be specified independently of any access patterns which are presented. In other words, the queue exhibits predictable performance characteristics. An

access requires a constant number of steps, with any shared data accesses requesting a constant number of words. For the cost analysis in Section 2, the values of w_i, h_i and n_i are constant. Asymptotically, the cost of a queue access is $\Theta(s+g+t+D)$. The *throughput* of the queue is the number of accesses carried out by p processors per unit time, compared to a sequential version. This can be formalised as $t(p,1)/t(p,p)$, where $t(n,p)$ is the time to perform n accesses on p processors. The throughput is thus $\Theta(p/\{s+g+t+D\})$. Assuming s, g and t are constant gives $\Theta(p/D)$, which matches the frequency f, stated in Section 4.1.

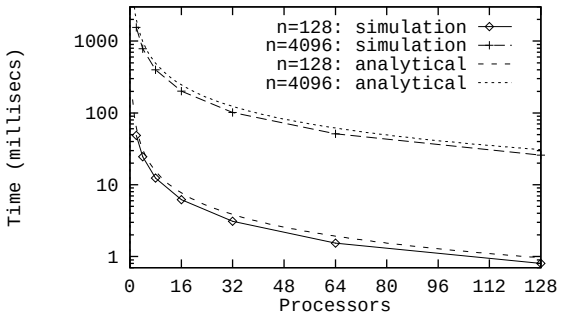
4.3 The practical performance of the queue

A discrete event simulation of the WPRAM model on a distributed memory machine has been written [6], based on the performance of the T9000 and C104 products [11]. It includes the ability to pipeline multiple shared data accesses and to overlap communication with local computation. A table of the dominant performance parameters and WPRAM operation costs is provided below. D is taken to be constant over the given range of processors, and the minor local computation is assumed to be dominated by shared data access costs. From these parameters, the cost of the queue operations can be derived (using the WPRAM code available on the simulator).

Machine Parameter	Cost
D	30 μsecs
g	5.6 μsecs per word

WPRAM Operation	Cost
Access x global shared words	$D+xg$
Tag operation	$D+g$

Queue Operation	Cost
Insert	$7D+9g = 260\ \mu secs$
Delete	$19D+24g = 704\ \mu secs$



A comparison of the simulation results and analytical analysis for the queue is presented in the above graph. In this example, each of the p processors inserts and deletes n/p queue items (shared address values). As can be seen, the results compare favourably. The analytical costs are higher since a worst-case costing is provided (assuming all options which lead to code execution are taken). For example, when p and n are both 128, simulation results give 802 $\mu secs$, as opposed to 964 $\mu secs$ by the analytical method.

5 Conclusions

This paper has presented a computational model, the WPRAM, for the design and analysis of scalable algorithms. The WPRAM extends the bulk synchronous approach of the BSP with operations to support more irregular forms of parallelism. Using the example of a shared queue data structure, it was shown how shared abstract data types SADTs) can be used to support tractable performance analysis, as well as being used to provide modular and portable code development. Current work is focusing on the use of SADTs in the design of scalable dynamic load balancing methods [8].

References

[1] M. E. Dyer, J. M. Nash, and P. M. Dew. An Optimal Randomized Planar Convex Hull Algorithm With Good Empirical Performance. In *The Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 21–26, 1995.

[2] K. Gharachorloo, S. V. Adve, A. Gupta, and J. H. Hennessy. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[3] M. Herlihy. Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15, 1993.

[4] W. F. McColl. An Architecture Independent Programming Model For Scalable Parallel Computing. In J. Ferrante and A. J. G. Hey, editors, *Portability and Performance for Parallel Processing*. John Wiley and Sons, 1993.

[5] J. M. Mellor-Crummey and M. L. Scott. Synchronisation Without Contention. In *ASPLOS*, volume 4, pages 269–278, 1991.

[6] J. M. Nash. *A Study of the XPRAM Model for Parallel Computing*. PhD thesis, School of Computer Studies, University of Leeds, 1993.

[7] J. M. Nash, P. M. Dew, J. R. Davy, and M. E. Dyer. Implementation Issues Relating to the WPRAM Model for Scalable Computing. In *EuroPar96*, pages 319–326. Springer Lecture Notes in Computer Science (volume I), 1996.

[8] J. M. Nash, P. M. Dew, J. R. Davy, and M. E. Dyer. Scalable Dynamic Load Balancing using a Highly Concurrent Shared Data Type. In *The 2nd European School of Computer Science: Parallel Programming Environments for High Performance Computing*, pages 123–128, April 1996.

[9] J. M. Nash, P. M. Dew, and M. E. Dyer. A Scalable Concurrent Queue on a Message Passing Machine. *The Computer Journal*, 39(6), 1996.

[10] J. M. Nash, M. E. Dyer, and P. M. Dew. Designing Practical Parallel Algorithms for Scalable Message Passing Machines. In *WTC'95 World Transputer Congress*, pages 529–544, 1995.

[11] P. W. Thompson and P. W. Welch. *Networks, Routers and Transputers*. IOS Press, 1994.

[12] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.