Empirical Study of Parallelism Throttling Schemes on a Massively Parallel System

Y.M. Teo and J.C. Yan
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore 0511
email: teoym@iscs.nus.sg

Abstract

Exploitation of parallelism in massively parallel systems is intuitively appealing and is a promising avenue for achieving teraflop performance. However, parallelism is not free; apart from overheads in communication and synchronisation, having too much program parallelism can also raise serious resource management problems during program execution. The problem of resource management is particularly complicated by the distributed nature of massively parallel systems.

In this paper, we address the issue of managing parallelism in a massively parallel system. This is an extension of our previous work on hardware throttle for parallel systems. We propose two parallelism throttle schemes, and a scheme for monitoring and measurement of system workload. Processes in the system are initiated when runtime loading levels in the system permit. The aim is to match dynamic program parallelism *to static* machine parallelism. *Experimental study is conducted using a simulated Multi-cluster Dataflow Machine as testbed. Our study shows the extent that control over runaway program parallelism is necessary, and that it is possible to have good distributed control of resource use in a massively parallel system.*

1 Introduction

Most of today's massively parallel computers are based on the data parallel model of computation by which the principal data structures of a problem are partitioned and assigned to the processors of the machine. Large-scale parallel computation where hundreds of processors execute distinct parts of a problem exploiting functional parallelism is becoming common. There appears to be widespread consensus that general purpose parallel computer of the future will be massively parallel architecture consisting of many processing nodes connected via a high speed and regular interconnection network. Massively parallel architectures are confronted with two main widely discussed fundamental problems: *memory latency problem* and *synchronisation problem* [4]. Current processing node design in general employs multithreaded parallelism to overlap communication and synchronisation latencies with computation to achieve better processor and network utilisation. Besides these issues, there are resource management related issues such as thread management, memory management, traffic management, load distribution and balancing, parallelism (or concurrency) management, *etc.*

A major source of parallelism in programs is algorithmic parallelism introduced by the programmer. The potential problem of too little program parallelism is clear. If there is too much parallelism, we are in danger of filling the entire memory of the machine with half-finished computations, none of which can proceed due to lack of space. Control of parallelism in parallel systems has been widely studied. Control can be introduced at different levels in a parallel system such as program annotations at the language level, strict execution of function and loop iteration can be enforced by the compiler, dynamic control can be exercised using LIFO/FIFO queues, priorities, *etc.* at the hardware level [8]. In the MIT Dataflow Machine, a hybrid method of control called the K-bounded loop [3] is used. Analysis of the resource required to support loop execution is done at compile-time and based on runtime workload, K-loops are unravelled for parallel execution.

Research so far has focused on strategies for parallel machines with limited machine parallelism. To the best of our knowledge, recent massively parallel architectures such as Alewife [1], Tera [2], *T [7] and MASA [6] do not address the issue of parallelism management. The effectiveness and scalability of parallelism throttle are important issues requiring attention in massively parallel systems. This paper addresses the parallelism management problem in massively parallel systems. The rest of this paper is organised as follows. In section 2, we discuss two schemes for dynamic control of program parallelism in a massively parallel system. The schemes are implemented on a Multi-cluster Dataflow Machine that exploits massive *spatial* and *temporal* parallelism. The performance of these two schemes are analysed in section 3. Section 4 describes the limitations in our approach. In conclusion, we summarise the main results and assess the importance of memory resource management in future massively parallel systems.

2 Strategies for Dynamic Parallelism Control

The basic strategy for controlling inter-process parallelism is to delay the execution of new processes unless machine resource permits. There are two main aspects in designing a throttle. Firstly, a

workload monitoring and measurement scheme is required to determine the activity level of the system. Secondly, a strategy is required to handle process activation request and to determine the appropriate process to start up. We define *parallelism throttling* as the task of matching dynamic *program parallelism* to *machine parallelism* at runtime. Static parallelism control schemes are discussed in [8]. Our throttle design is guided by two objectives: keep all processors as busy as possible and introduce minimal throttle overheads.

We propose two control strategies: centralised or distributed control. In the former case a *global resource manager* is responsible for selecting and dispatching processes to idle processors. In the latter case no single resource manager makes global decisions for initiating new processes, but the processors themselves are responsible for determining which process to execute next. Using a Multi-cluster Dataflow Machine as testbed, the architecture of the two control schemes are discussed in the sections below. The architecture of a cluster is similar in configuration to the Multi-ring Manchester Dataflow Machine [5], and consists of a number of processing element (PE) rings and structure store (SS) rings interconnected by a multi-stage switch. A Multi-cluster Dataflow Machine in turn consists of many clusters interconnected by another multi-stage switch called the global communication switch. The processing element and structure store have similar architecture to the Manchester Dataflow Machine [5].

2.1 Centralised Parallelism Throttle Scheme

The architecture of a massively parallel system with centralised throttle control is shown in figure 1. A *global resource manager (GRM)* connected to the

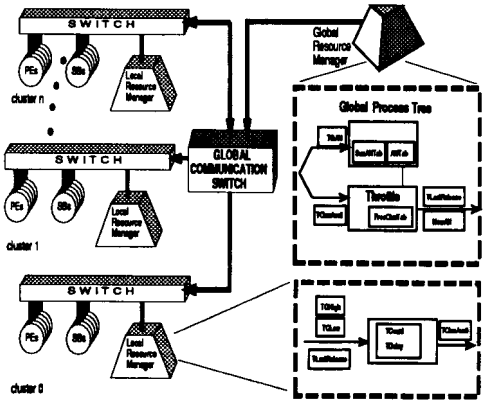

Figure 1: Massively Parallel System with Centralised Throttle Control

global switch keeps track of process execution. The tasks of workload monitoring, data structure management are performed by a *local resource manager (LRM)* in each cluster. Before a process can execute, a startup process request is sent to the GRM and suspended in a *global process tree (GPT)* data structure. Based on runtime cluster activity measures, the GRM allocates processes to various clusters for execution.

2.1.1 Cluster Local Workload Measurement

Workload is monitored and measured at the cluster level similar to the scheme proposed in [8, 9]. Each PE's token queue (TQ) unit keep tracks of its own TQ size and maintains its own logical bands/hysteresis. When a token queue band boundary is exceeded, it sends a high/low message token to its LRM. A high message token (*TQHigh*) increments a counter, *TCount*, while a low message token (*TQLow*) decrements this counter. TCount is used as a measure of the activity level in a cluster. The algorithm is shown in figure 2. When the

```
Token_Queue Unit:
  current-band = 1
  if TQ length > (current-band * bandsize + hysteresis) then
    send TQHigh to Local_Resource_Manager
    current-band = current-band + 1
  if TQ length < ((current-band - 1) * bandsize − hysteresis) then
    send TQLow to Local_Resource_Manager
    current-band = current-band − 1

Local_Resource_Manager:
  case message-token of
    TQHigh : TCount = TCount + 1
    TQLow : TCount = TCount − 1
  TDelay = ⌊(k * (e^(TCount/p) − 1))/p⌋
  if (suspend-process > 0) and
    (last-release-time + TDelay + 2 * switch_delay)
      <= current-time) then
    send TClusAvail to Global_Resource_Manager
```

Figure 2: Parallelism Throttle - Centralised

LRM deems the cluster free enough through periodic sampling of TCount and calculation of *TDelay*, a *TClusAvail* token is sent to GRM. *TDelay* is a heuristic function that estimates the amount of time the cluster takes to complete execution of allocated processes. Variable k is a constant, and p denotes the number of processing elements in a cluster. At the global level (see figure 1), the GRM maintains a table (FreeClusTab) indicating which clusters are free to accept additional tasks. Upon the release of a task to a cluster, the corresponding flag would be set to indicate that the cluster is busy. This flag will be cleared subsequently when a TClusAvail is received from that cluster's LRM.

2.1.2 Operation

Initially, all clusters are free and requests for process startup are serviced immediately. Within each cluster, token queue lengths are initialised to zero and as such, the LRM will swiftly send TClusAvail tokens to the GRM to initiate new processes. This achieves our aim of rapidly starting new tasks to exploit the processing capacity of all nodes. The latency between startup request and actual startup at this stage is at a minimum equal to the travel time of the message tokens from LRM to GRM and back. Therefore, in estimating the optimal moment to initiate a new process in its cluster (figure 2), the LRM factors in the time of $2 * switch\_delay$ to compensate for this latency.

The GRM on receiving the TClusAvail token selects the *leftmost-deepest process* from the GPT for execution on the requesting cluster. This strategy of releasing processes gives priority to child processes belonging to processes that have already started execution. The aim is to reduce the lifetime of a process which in turn has the effect of reducing the amount of memory required.

2.2 Distributed Parallelism Throttle Scheme

As more clusters are added, the GRM in the centralised control scheme becomes a bottleneck. We discard the idea of having a central body with all the load and process execution information in favour of a distributed approach to resource management. This will alleviate the GRM bottleneck but our problem becomes one of how to let each cluster, with only local information, make effective process scheduling decisions. The architecture of a machine with distributed throttle control is shown in figure 3. A

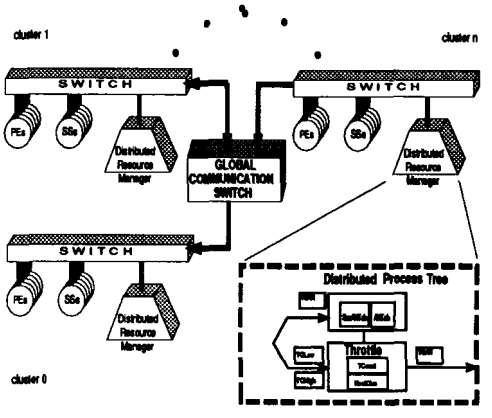

Figure 3: Massively Parallel System with Distributed Throttle Control

distributed resource manager (DRM) in each cluster performs control of parallelism within the cluster. The *global process tree (GPT)* is distributed among the DRMs. Each DRM maintains a subtree of the GPT call *distributed process tree (DPT)*. The workload monitoring scheme used in the centralised throttle case is adopted.

Process startup requests are sent to the DRM instead of the GRM. Process distribution amongst the clusters is performed in an asynchronous round-robin manner as discussed in section 2.2.1. The DRM in the originating cluster determines the target cluster to execute new process. Hence, every cluster will parcel out work (process) to every other cluster in turn, including itself. Process startup requests received from other clusters are suspended at the DPT. Processes are released when cluster activity level permits using the same heuristic as in the centralised case. Processes are released within every cluster independent of each other. This allows concurrent process release scheduling.

2.2.1 Distributed Process Scheduling

The process distribution scheme used in distributed throttle control is illustrated using a task graph in figure 4. The graph shows the schedule of

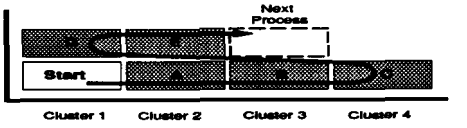

Figure 4: Brick-laying Process Distribution

assignment of new processes in a four cluster system from the viewpoint of cluster one (referred to as the *reference cluster*). Each block represents a new process allocated by the reference cluster. The round-robin process distribution has a "brick-laying effect" that distributes processes very quickly to every cluster. The task graphs in figure 5 show the process distribution in the system from the perspective of each cluster in turn. Each cluster spawns

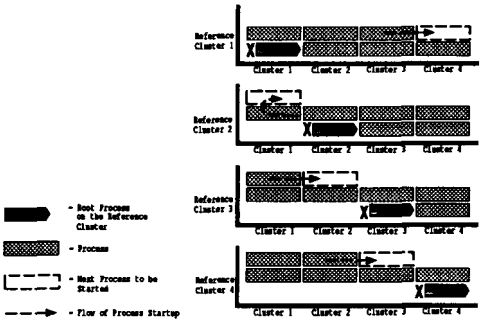

Figure 5: Process Distribution in a Four Cluster System

seven processes and brick-layers them from the adjacent cluster onwards. Starting process in a reference cluster is denoted by "x".

We make two observations when determining the worst load imbalance. The first is that each reference cluster will be fair in its allocation to within one process. This is obvious from the "brick-laying" principle. The second observation is that the most imbalance occurs when all reference clusters penalise the same cluster in allocating their processes. This happens if all clusters give an additional process to the same cluster, then that cluster will be the busiest by *n-1* processes. The converse is true if all clusters short change the same cluster and make it the freest by *n-1* processes. These two worst case imbalanced distributions are shown in figure 6. The difference in the number of processes allocated to a cluster is never greater than the total number of clusters. We can also generalise that worst case distribution is always skewed diagonally with the first cluster, having one more process than the rest.

3 Simulation Results and Discussion

The two throttling schemes discussed were implemented on a simulated Multi-cluster Dataflow Ma-

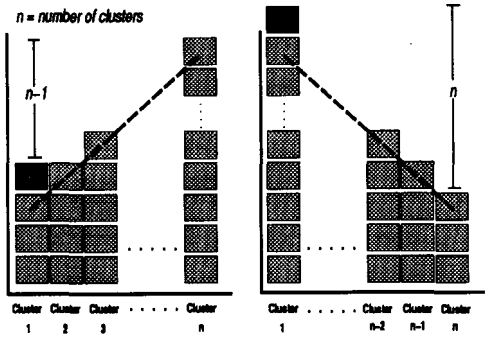

Figure 6: General Worst Case Load Imbalances

chine. The simulator is time-driven, each operation performed during a run is time-stamped. To gauge the amount of memory used, the sum of tokens in all token queues and matching stores are sampled. The average utilisation of processors denoted by *%PU* is also captured. The number of additional tokens used for throttling divided by the total number of tokens used during a run is used to calculate the overhead of throttling (denoted by *%ov*). This would include the TQHigh and TQLow tokens.

To measure the scalability of the throttles, simulations were performed using BINTEG, a recursive area subdivision program written in SISAL which produces 2^n processes given the parameter n. By adjusting n, we could present a constant amount of workload per cluster whilst the number of clusters in the machine is varied. As figure 7 shows, on a two-cluster machine with 4 PEs and 4 SSs per cluster running BINTEG(n=11), the centralised throttling scheme reduced memory usage by more than 90%. Under similar conditions, using a distributed

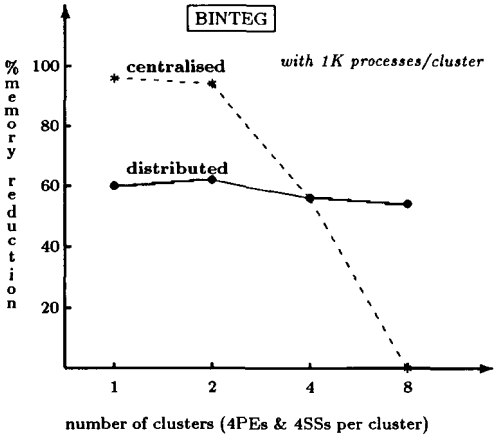

Figure 7: Throttle Scalability - With Constant Workload per Cluster

throttling scheme, a reduction of more than 60% was achieved. When the number of clusters was increased, keeping the workload per cluster constant, the percentage memory reduction achieved with centralised throttling dropped to zero while the distributed system stays at a near constant level of around 60%. Table 1 gives a more detailed breakdown of the amount of memory used (measured in terms of number of tokens in various queues) during runs of BINTEG(n=13). Together with informa-

no. of clusters	*centralised*		*distributed*	
	unthrottled	throttled	unthrottled	throttled
4	22546	7734	51688	20460
8	20663	20661	52263	24198
12	20548	20548	52255	27327
16	20251	20751	52521	28479
24	20489	20489	48345	30704
32	21016	21016	48113	34884

Table 1: BINTEG(n=13) - Total Memory Usage

tion gathered on switch occupancy and input queue lengths at each module, it is clear that the centralised throttle is not responsive enough to keep a large system fully occupied. Restriction of memory use by the bottleneck at the GRM is ironically extremely effective, so much so that throttling becomes inconsequential.

The result of simulations using BINTEG(n=13) also showed good linearity in utilisation in relation to the number of clusters using the distributed scheme. This is shown in figure 8.

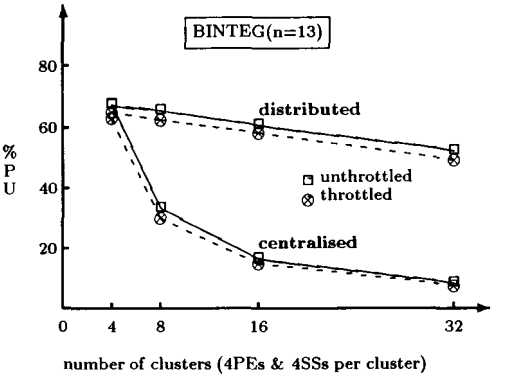

Figure 8: Average Processing Unit Utilisation

The percentage utilisation and throttle overhead together with the number of timesteps (ts) required in each run for a workload of 1024 processes per cluster is shown in table 2. We are therefore optimistic that such a distributed throttling system can reduce the amount of memory used by a massively parallel machine, keeping it at a near constant level of activity. We can further optimise the machine by

no. of clusters	centralised			distributed		
	%PU	ts	%ov	%PU	ts	%ov
1	71.3	64183	0.56	72.9	62861	1.24
2	69.3	66029	0.55	69.4	65914	1.21
4	61.3	74350	0.25	63.6	71700	1.13
8	30.0	149100	0.0	62.4	73100	1.12

Table 2: Throttle Scalability - With Constant Workload per Cluster

modifying the eagerness with which we restrict resource use. This can be accomplished by tuning the parameters within the throttles.

3.1 Varying Machine Configurations

Simulation results for programs BINTEG and NQUEENS are shown in figures 9 and 10. The

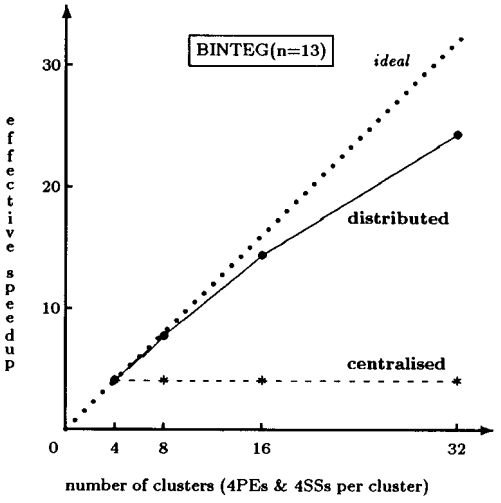

Figure 9: Parallelism Throttle: Speedup for BINTEG(n=13)

NQUEENS program works out the solution for placing n queens on a n-by-n chess board in such a way that no queen checks any other queens. The search for solutions is performed using a parallel divide-and-conquer algorithm. The program problem size selected has sufficient program parallelism to fully saturate the machine. Distributed throttling shows good speedup in performance up to 32 cluster (128 PE) sized machine.

4 Limitations

There are two related aspects of scheduling; parallelism control and load balancing. The former has to do with the reduction of excessive runtime program parallelism while the latter is more concerned with ensuring that processes are started in clusters which are relatively free. We have focused our attention on the former since the latter is more of an

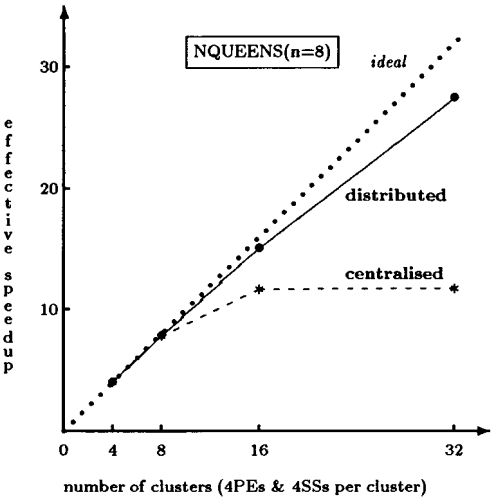

Figure 10: Parallelism Throttle: Speedup for NQUEENS(n=8)

efficiency consideration. A significant shortcoming of distributed round-robin scheduling is that it does not explicitly handle the load balancing issue. Allocation of work to each cluster does not take into account the current load on that cluster. Thus at a particular moment when one cluster is free and another is busy, a new process may be queued for execution at the busy cluster instead. We rely on the throttle at the busy cluster to regulate resource use internally, but it would probably be more desirable to send the activation request to the free cluster. Fortunately, this problem becomes minimal when a sufficiently large number of processes are generated during a run, as is usually the case with large scale problems. Under such circumstances, having nearly the same overall number of processes run in each cluster (a fact assured by round-robin scheduling) guarantees, to a certain extent, fairness in load distribution. One can draw an analogy with selecting grains of sand (processes) and filling a number of sacks (clusters) one by one. It matters not that the size of each grain may differ considerably, the resulting sacks would be pretty much of the same size given a sufficiently large number of grains per sack.

5 Conclusions

Two throttling schemes implemented on a simulated massively parallel machine are discussed. We show that runaway program parallelism introduces unnecessary communication traffic and is detrimental to system performance. It is observed that the distributed throttle scheme is more effective in restraining program parallelism than the centralised scheme when program parallelism is greater than machine parallelism, and also vice-versa. Distributed control enables each cluster to respond and to react to fluctuations in dynamic program parallelism more readily. Centralisation of process control

increases the latency of initiating process and introduces a serious traffic bottleneck at the global resource manager. Experimental results indicate that detailed measurement of machine loading level in massively parallel systems, which can be expensive, is not critical. The approximated load measuring scheme used is sufficient for effective parallelism control. Simulation experiments varying the number of clusters from four to thirty-two show good speedup and demonstrate the scalability of the distributed throttle scheme. Memory utilisation reduction of more than 50%, and throttle overhead of less than 1.5% (measured in terms of additional tokens introduced divided by the total number of tokens) demonstrate the effectiveness of the parallelism throttle.

Parallelism throttling is an important aspect of massively parallel system design if machine performance is to be maximised and system resources are to be effectively utilised. In a multiprogramming environment, parallelism control can increase the number of programs executed at the same time. However, when the minimum amount of memory required to execute a program exceeds available system memory, program execution cannot proceed. In this instance, parallelism control is of no help and virtual memory is required [10]. Virtual memory and parallelism throttle mechanisms are essential features of practical massively parallel systems of the future.

Acknowledgements

Special thanks go to Kee Chee Keong and Lee Siew Leng for their contributions to the work carried out. The work reported in this paper was supported by the National University of Singapore under grant RP900629.

References

[1] A. Agarwal, et al., *"The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor"*, in Scalable Shared Memory Multiprocessors, edited by M. Dubois and S. Thakkar, pp. 239-261, Kluwer Academic Publishers, 1992.

[2] R. Alverson., et al., *"The Tera Computer System"*, Proc. International Conference on Supercomputing, pp. 1-6, Amsterdam, 1990.

[3] Arvind and D.E. Culler, *"Managing Resources in a Parallel Machine"*, in Fifth Generation Computer Architectures, J.V. Wood (ed.), Elsevier Science Publishers B.V. (North-Holland), IFIP, pp. 103-121, 1986.

[4] Arvind and R.A. Iannucci, *"Two Fundamental Issues in Multiprocessing"*, Proc. DFVLR - 1987 Conference on Parallel Processing in Science and Engineering, West Germany, Springer-Verlag LNCS 295, June 1987.

[5] J.R. Gurd, C.C. Kirkham and I. Watson, *"The Manchester Prototype Dataflow Computer"*, CACM, vol. 28, no. 1, pp. 34-52, January 1985.

[6] Jr R.H. Hilstead and T. Fujita, *"MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing"*, Proc. 15th Annual Symposium on Computer Architecture, pp. 443-451, USA, 1988

[7] R.S. Nikhil, G.M. Papadopoulos and Arvind, *"*T: A Multithreaded Massively Parallel Architecture"*, Proc. of 19th Annual Symposium on Computer Architecture, pp. 156-167, Australia, 1992.

[8] Y.M. Teo, *"Managing Parallelism in a Parallel Computer"*, in Frontiers in Parallel Computing, edited by V.P. Bhatkar, Centre for Development of Advanced Computing, India, pp. 3-20, 1991.

[9] Y.M. Teo and A.P.W. Bohm, *"Resource Management and Iterative Instructions"*, in Advanced Topics in Dataflow Computing, edited by J.L. Gaudiot ad L. Bic, Chapter 18, pp. 481-499, Prentice-Hall, 1991.

[10] Y.M. Teo and I.W. Chan, *"Virtual Memory in a Dataflow Computer"*, in Parallel Programming Systems, edited by C.K. Yuen and Y. Yonezawa, pp. 137-147, World Scientific, 1993.