Designing Scalable Object Oriented Parallel Applications

João Luís Sobral, Alberto José Proença

Departamento de Informática - Universidade do Minho
4710 - 057 Braga – Portugal
{jls, aproenca}@di.uminho.pt

Abstract The SCOOPP (Scalable Object Oriented Parallel Programming) system efficiently adapts, at run-time, an object oriented parallel application to any distributed memory system. It extracts as much parallelism as possible at compile time, and it removes excess of parallel tasks and messages through run-time packing. These object and call aggregation techniques are briefly presented. A design methodology was developed for three main types of scalable applications: pipeline, divide & conquer and farming. This paper reviews how the method can help programmers to design portable and efficient parallel applications. It details its application to a farming case study (image threshold) with measured performance data, and compares with programmer's tuned versions in a Pentium cluster.

1 Introduction

The development of portable parallel applications that *efficiently* run on several platforms imposes a platform-tuning overhead. This paper addresses issues that may reduce this overhead, namely to up/down scale an object oriented parallel application, including the automatically tuning of the application for each platform.

Applications may require dynamic granularity control to get an acceptable parallel execution performance in time-shared platforms, namely when the parallel tasks are dynamically created and whose behaviour cannot be accurately estimated at compile-time. Few systems provide dynamic granularity control [1][2][3][4], based on fork/join parallelism constructs, ignoring the fork construct and executing tasks sequentially (parallelism serialisation).

The SCOOPP system [5] is a hybrid compile and run-time system, that extracts parallelism, supports explicit parallelism and dynamically serialises parallel tasks and packs communication in excess at run-time. A design methodology was developed for three main types of scalable applications: object pipelines [6], static object trees (e.g., farming) and dynamic object trees (e.g., divide & conquer). Several case studies have been tested on various platforms: a 7 node Pentium cluster, a 16 node PowerPC based Parsytec PowerXplorer and a 56 node Transputer based MC-3.

This paper shows evaluation results that were experimentally obtained by executing a farm type application and a comparison with a programmer's optimised

[*] This work was partially supported by grant PRAXIS 2/2.1/TIT/1557/95

version, on a 7 node Pentium III based cluster, under Linux with a threaded PVM on TCP/IP. The cluster nodes are inter-connected through a 1,2 Gbit Myrinet switch.

Section 2 presents an overview of the SCOOPP system. Section 3 introduces a farm type application, the design methodology applied for scalability, and presents performance results. Section 4 closes the paper with suggestions for future work.

2 SCOOPP System Overview

SCOOPP system scales parallel applications to any distributed memory systems, in two steps: at compile-time, the compiler and/or the programmer specifies a large number of fine-grained parallel tasks; at run-time, parallel tasks are packed into larger grains - according to the application/platform behaviour and based on security and performance issues – and communications are packed into larger messages.

The SCOOPP programming model is based on an OO paradigm supporting both active and passive objects. Active objects (parallel objects in SCOOPP) specify explicit parallelism: they model parallel tasks, they may be placed at remote processing nodes and they communicate through either asynchronous or synchronous method calls. Passive objects take advantage of existing code; they are placed in the context of the parallel object that created them, and only copies can be moved between parallel objects; method calls on these objects are always synchronous.

SCOOPP extracts parallelism by transforming selected passive objects into parallel ones [7], and at run-time it removes parallelism overheads by transforming (packing) parallel objects in passive ones and by aggregating method calls [8].

These run-time optimisations are implemented through:

- method call aggregation: (delay and) combine a series of asynchronous method calls into a single aggregate call message; this reduces message overheads and per-message latency;
- object agglomeration: when a new object is created, create it locally so that its subsequent (asynchronous parallel) method invocations are actually executed synchronously and serially.

The decision to pack objects and method calls considers several factors: the latency of a remote "null-method" call (λ), the inter-node communication bandwidth, the average overhead of the method parameters passing (ν) and the average local method execution time on each node type (ε). More details of the run-time granularity control and how decision factors are estimated can be found in [6].

3 Design and Performance Evaluation of Farming Applications

This section presents and analyses a farming parallel algorithm, based on master and slaves. The master executes the sequential part of the work, e.g., it divides the work into several tasks, sends them to the slaves and joins the received processed results.

In SCOOPP, two parallel objects classes implement the farming applications: master and the slave classes. Both classes have other methods, which promote code reuse, since the master and slave classes are generic.

The design of a scalable farming application to efficiently run on several target platforms, must adequately address three main issues: the number of slaves to specify, the master hierarchy (if any), and the task granularity.

A high number of slaves helps to scale the application to a larger system, since the number of slaves limits the number of nodes that the application can use. However, if the slave/node is high, performance may suffer due to the slave management time. Specifying a number of slaves equal to the number of nodes limits dynamic changes on the number nodes used and when more than one master is used, it may be easier to use a number of slaves proportional to the number of masters.

Using just one master may limit the application performance, since the tasks and slaves management is centralised. A high number of slaves should be followed by a decentralisation of management work, by using a master hierarchy. However, using several masters introduces overheads due to the coordination among masters.

The specification of the task size depends on the number of slaves and on the target platform. The work division should provide a number of tasks higher than the number of slaves to provide enough work for all slaves. A high number of tasks helps the load distribution, but also introduces higher overheads, due to the additional work to join and split work, and each task may be too small for a platform.

The SCOOPP methodology can help to achieve an adequate solution to these issues, showing the feasibility to develop parallel applications, that are portable and scalable on several target platforms, without requiring source code changes. The SCOOPP methodology suggests the programmer to specify a high number of slaves and masters (e.g., parallel objects) and a high number of parallels tasks (e.g., method invocations). The SCOOPP run-time system is responsible to pack excess of masters and slaves, and to aggregate method invocations, reducing the impact of the overhead due to excess of parallelism and communication.

3.1 Packing Policy for Farming Applications

Packing policies defines “when” and “how much” to pack. These are grouped according to the structure of the application: object pipelines, static object trees and dynamic object trees. This section focuses on packing policies for farming.

When communication overhead becomes higher than the task processing time, method calls should be packed. This occurs in SCOOPP when the overhead of a remote method call - given by the sum of the average latency of a remote "null-method" call (λ) and the overhead of the method parameters passing (ν) is higher than the average method execution time, ε, e.g., $(\lambda+\nu)>\varepsilon$. This is the turnover point to pack, where the communication overhead is considered too high.

The communication grain-size, G_m, is computed from the λ, ν and ε and defines “how many” method calls to pack into a single message. Sending a message that packs G_m method calls introduces a time overhead of $(\lambda+G_m\nu)$ and the time to execute this pack is $G_m\varepsilon$. Packing should ensure that $(\lambda+G_m\nu)<G_m\varepsilon$, e.g., that the overhead to send a pack of several remote method calls is lower than the time to locally execute the pack of method calls, e.g., $G_m=\lambda/(\varepsilon-\nu)$. This point defines the minimum number of method calls to pack into a message. The packing policy is

based on a dynamic criterion, which initially packs λ/(ε-ν) method calls on each message, to spread the work rapidly, and progressively increases the number of method calls on each pack, proportionally to the number of packs sent to each node (μ), to further reduce the overheads, e.g., Gm=λ(1+μ)/(ε-ν).

Object farming places some limitations on object packing. When only one master is provided, the excess of parallel objects (slaves) cannot be effectively removed, since to remove parallelism they should be packed with the master, which concentrates most of the slaves on the master's node. Note that packing slaves together do not remove parallelism overheads, since no method calls are performed between slaves. However, this limitation can be overcome by using a hierarchy of masters, which allows the packing of slaves with intermediate masters.

3.2 A Farming Case Study

An image processing case study, using a farming structure, was selected to show the impact of SCOOPP: a dynamic threshold with a predefined window size. The application was developed on a C++ SCOOPP prototype [7][8]. Performance results were obtained using an image of 512x512 (windows of 11x11) and 16 slaves (~2/node), and a 2-level master hierarchy.

In this application the master splits the image into frames and sends the frames to the slaves (call a method on the slaves). Each slave processes the frames and sends them back to the master (call a method on the master). On this case study, frames are square-shaped and there is some frame overlap to reduce intermediate communication during frame processing.

To select the frame size, a programmer may opt for a higher number of frames, if she/he aims scalability for large distributed systems. To guarantee an efficient execution, the programmer has to tune the application for each target platform. Fig.1 shows the several experiments data she/he had to collect (using 4096 frames and manually packing several frames per message), just for two platforms: a 4- and a 7-node cluster. Using SCOOPP, a single point is automatically obtained for any platform, where performance is very close to the optimum value.

Table 1 shows how the grain-size of the selected frame may affect the overall performance, without any further tuning (worst case). This table clearly suggests that

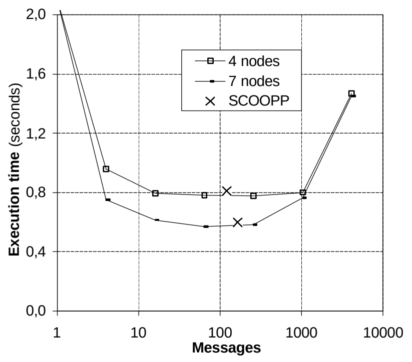

Fig. 1. Execution times using a single master

Table 1. Execution times with and without SCOOPP and decision factors (7-nodes).

Frames	Programmer	SCOOPP	Gm	ε	λ+ν
1	2,11	2,10	1	672k	54k
4	0,57	0,56	1	218k	14k
16	0,45	0,45	1	60k	4129
64	0,39	0,39	1	15k	1415
256	0,36	0,37	2	4k	673
1024	0,49	0,40	6	997	455
4096	**1,45**	**0,56**	**29**	**260**	**385**
16384	5,20	1,07	95	66	329

the use of 256 frames might be the best option for this case study and target platform (it still requires manual tuning), and that the programmer should test several parameters for each platform for an adequate tuning. However in a larger and time shared environment, the use of 4096 frames may prove an advantage. By applying the SCOOPP methodology, significant gains in performance can be obtained.

Conclusion

The success of parallel computation in the past did not had the expected results mainly due to (i) the lack of adequate tools to support automatic mapping of the applications into distinct target platforms, without compromising efficiency, and (ii) the portability costs due to excessive overhead to tune the application for each target platform. Current trends (time-shared clusters and the Grid) place additional challenges, namely on dynamic tuning. SCOOPP attempts to overcome these limitations by providing dynamic and efficient scalability of object oriented parallel applications across several target platforms, without requiring any code modification. The presented results show the effectiveness of the SCOOPP methodology when applied to farming applications; it dynamically increases grain-sizes, improving execution times and showing that this methodology successively identifies and removes most parallelism overheads.

Current experimental results gathered data related to both the platform and the application behaviour; the latter is dynamically updated, while the former was, so far, statically gathered before execution. Research work is also being carried out to support a dynamic decision mechanism based on stochastic approaches [9].

References

1. Mohr, E., Kranz, A., Halstead, R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, IEEE Trans. on Par.& Dist. Proc., Vol. 2(3), July (1991)
2. Goldstien, S., Schauser, K., Culler, D: Lazy Threads: Implementing a Fast Parallel Call, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)
3. Karamcheti, V., Plevyak, J., Chien, A.: Runtime Mechanisms for Efficient Dynamic Multithreading, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)
4. Taura, K., Yonezawa, A.: Fine-Grained Multithreading with Minimal Compiler Support, Proc. ACM SIGPLAN CPLDI'97, Las Vegas, July (1997)
5. Sobral, J., Proença, A.: Dynamic Grain-Size Adaptation on Object-Oriented Parallel Programming - The SCOOPP Approach, Proc. 2nd IPPS/SPDP, Puerto Rico, April (1999)
6. Sobral, J., Proença, A.: A SCOOPP Evaluation on Packing Parallel Objects in Run-time, VecPar'2000, Porto, Portugal, June (2000)
7. Sobral, J., Proença, A.: ParC++: A Simple Extension of C++ to Parallel Systems, Proc. of the 6th Euromicro Work. on Par. & Dist. App. (PDP'98), Madrid, Spain, January (1998)
8. Sobral, J., Proença, A.: A Run-time System for Dynamic Grain Packing, Proceedings of the 5th Int. EuroPar Conference (Euro-Par'99), Toulouse, France, September (1999)
9. Santos, L., Proença, A.: A Bayesian RunTime Load Manager on a Shared Cluster, Scheduling and Load Balancing on Clusters (SLAB'2001), special session in IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid'2001), Brisbane, Australia, May, 2001