Dynamic Grain-Size Adaptation on Object Oriented Parallel Programming The SCOOPP Approach

João Luís Sobral, Alberto José Proença
Departamento de Informática, Universidade do Minho, 4710 Braga, PORTUGAL
[jls, aproenca]@di.uminho.pt

Abstract

This paper presents the SCOOPP (SCalable Object Oriented Parallel Programming) approach to support the design and execution of scalable parallel applications.

The SCOOPP programming model aims the portability, dynamic scalability and efficiency of parallel applications. The SCOOPP is an hybrid compile and run-time system, which can perform parallelism extraction, supports explicit parallelism and performs dynamic granularity control at run-time.

The mechanism that supports dynamic grain-size adaptation is presented and performance evaluated on two parallel systems. The measured results show the feasibility of the proposed dynamic grain-size adaptation and a scalability improvement of parallel applications over static parallel OO environments, which suggests cost benefits to develop scalable parallel applications to run on multiple platforms

1. Introduction

The development of parallel applications often requires a performance tuning to each platform as a natural programming step, and/or using dedicated libraries to access system key facilities.

As parallel programming tools are progressively being widely adopted, parallel applications become less platform dependent. PVM and MPI also helped to mix applications developed under different programming environments, and the portability of applications is further pushed by Java.

Programmers are now required to develop applications using the policy *build once, run anywhere*. The design of parallel applications should address both types of portability issues: at code level and at performance level, i.e. the execution performance should be high on a wide range of actual or future platforms.

This work was partially supported by the SETNA-ParComp project (Scalable Environments, Tools and Numerical Algorithms in Parallel Computing), under PRAXIS XXI funding (Ref. 2/2.1/TIT/1557/95).

A key factor affecting the performance of a parallel application on a target platform is the parallelism granularity, considered here as the task computation/ communication ratio, and often referred to as grain-size. When designing a parallel application, the parallelism granularity is a critical issue, both at the algorithm level and at the target platform. An inadequate match of these two granularities can degrade performance.

An efficient use of the available resources on a parallel system requires enough grains (parallel tasks) to spread over the processors, to allow the execution of parallel tasks, i.e., the number of the computational grains should be larger than the number of parallel resources. This is usually accomplished at compile-time, extracting as much parallelism as possible, to complement the programmer defined parallel tasks. As a consequence of these finer grained tasks, more control overhead is required and the overall system performance may degrade due to excess of traffic messages.

The granularity of the parallel tasks should be larger than the granularity the target platform supports, minimising the task overheads and inter-tasks communication over computing time. To reduce the unwanted overhead, a limit should be placed on the number of the running grains per processor, which depends on both the application and the target platform. One way to limit the number of resident grains, and to reduce the excess parallelism is to transform some parallel tasks into sequential ones, i.e., to increase the grain-size of each parallel task, by packing tasks into the same grain.

Some of the most obvious advantages of packed grains over a large number of non-packed grains include: faster calls (intra-grain procedure calls versus inter-grains IPC, and no context switching) and faster process management (smaller number of processes/processor, requiring less overhead on task creation/destruction). The experimental results on this communication validate these advantages.

Programmer based grain-size control puts a strong burden on programmer activity either by the development of alternative algorithms and strategies to match each platform, or by adding programmer explicit mechanisms for grain-size control which decrease the algorithm clarity. These alternatives may require a deep knowledge of both the algorithm and the target platform: they are strongly platform dependent and in many cases there is not enough available information to define the correct granularity.

Static grain-size determination falls short on shared systems, where the available resources can only be measured when the application starts running; also, in most parallel applications the computational cost of each parallel task can only be accurately measured at run-time.

Dynamic granularity control through grain packing is one of the key features in the SCOOPP system, using the available run-time data to take decisions on the size of the grains, by packing parallel tasks into single grains, whenever appropriate.

The SCOOPP system is a step forward in the development of techniques for dynamic granularity control, applied to parallel OO languages. It is an hybrid compile and run-time system, which can perform parallelism extraction, it supports explicit parallelism and it performs dynamic granularity control at run-time. This paper stresses the run-time dynamic granularity control (more details of the SCOOPP system in [1]).

Section 2 references related work on grain packing, while section 3 and 4 present the SCOOPP programming model and the approach to grain-size control. Section 5 introduces the current prototype and section 6 evaluates some performance results; section 7 concludes with suggestions for future work.

2. Related work on grain packing

Most work performed on grain-size adaptation is based on static grain packing [2]. The key concept is to find and pack the tasks belonging to the critical path of an algorithm in a single grain, and perform the task partition for the reminder tasks based on the critical path; other approaches attempt to minimise the inter-tasks communication delays [3]. Both are based on a DAG (Direct Acyclic task Graph), built at compile-time, which limits its application to fine grain tasks or tasks with known behaviour. The parallel OO language Ellie [4] includes static grain-size adaptation.

The computational regularity in time and space of many data parallel algorithms makes them good candidates for automatic grain-size adaptation. Packing the processing required by various data items on a single task and packing various data requests in single messages increases the grain-size. Usually this packing is performed at compile-time, guided by the number of processors of the target platform. HPF, C** [5] and HPC++ [6] are languages which include these features. CHAOS [7] supports irregular data structures and performs communication scheduling and message packing at run-time, although it does not provide mechanisms to automatically control the granularity of the parallel tasks.

Dynamic grain-size adaptation has been applied to functional [8] and logic programming environments [9]. At run-time, each new parallel computation may originate a new parallel task or it can be computed by the current task, providing a way to increase the grain-size. Their effectiveness on shared memory systems has been proved; however, they highly rely on shared data structures and are based on DAGs.

3. The SCOOPP programming model

Parallel objects in SCOOPP are computationally autonomous and they model parallel tasks, behaving as processes in the CSP model. Parallel objects continually receive messages and execute the associated method, leading to inter-object concurrency. To achieve intra-object concurrency (internal concurrency), parallel objects may concurrently execute constant methods, i.e., those that do not change the state of the object.

The SCOOPP programming model also supports sequential objects; these objects are passive entities and behave like objects in conventional OO environments. Each sequential object belongs to the context of the parallel object which created it; only a copy of that object can move between contexts of parallel objects. Figure 1a presents an example of the relationship among parallel objects, sequential objects, parallel objects contexts and parallel tasks. Multiple parallel tasks may be triggered in a parallel object, if it has internal concurrency; otherwise a parallel object corresponds to a single parallel task.

References to parallel objects may be copied or sent as a method argument, which may lead to cycles in a dependence graph (a graph showing the possible references between objects, like the ones presented in Figure 1). The application's dependence graph becomes a DAG when this feature is not used.

Parallelism extraction may be performed by transforming user specified sequential objects into parallel objects, which increases the overall number of parallel tasks (Figure 1b) and decreases de grain-size.

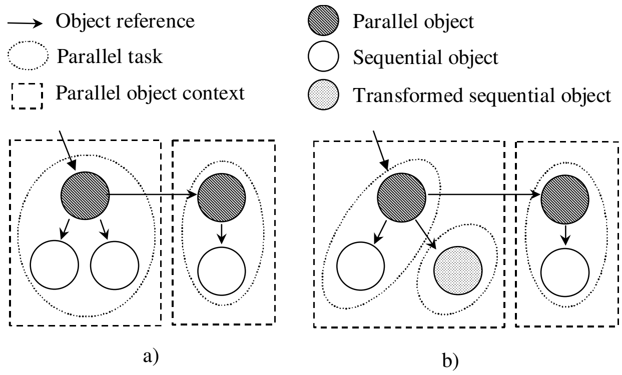

Figure 1. Parallel and sequential objects: a) without parallelism extraction; b) with parallelism extraction

Parallel objects may be composed of other parallel objects or inherit from other(s) parallel object(s). These features provide support to share or re-use code through component composition and refinement mechanisms.

Interactions with sequential objects are always synchronously executed, i.e. the caller waits until the method has been completed, however, interactions with parallel objects (and transformed sequential objects) may be synchronous or asynchronous. Method calls that do not return a value are asynchronously executed, i.e. the caller may proceed its own computation in parallel with the requested method execution. Method calls that return a value or that may raise exceptions are synchronously executed. This behaviour may be overridden by using explicit futures and specifying a repository for the method's result. That repository may be later consulted to check if the method has completed or to retrieve the result.

4. Dynamic grain-size in SCOOPP

To design a dynamic grain-size adaptation scheme three main questions are raised:

- How to adapt the grain-size?
- When to adapt the grain-size?
- Which tasks to pack/unpack?

This communication focus on the first issue and provides some leads to the second and third issues.

Most OO languages associate threads and/or processes to each object or method call, which usually leads to an undetermined number of grains (parallel tasks) of fixed grain-size: Charm++ [10] *chare* objects, UC++ [11] active objects, *mentat* and functional objects in Mentat [13] and parallel methods in COOL [12].

Dynamic grain adaptation in SCOOPP is accomplished in two phases: a static (compile-time) parallelism extraction [1], where a larger number of fine grains are created and/or marked, and a dynamic (run-time) grain control, through grain packing.

Grains are packed in the SCOOPP system by a controlled dissociation of threads/processes from methods/objects and by limiting the internal concurrency on parallel objects (i.e., number of concurrently executing methods). A single thread/process may be bind to a set of parallel objects, forming one grain (a single parallel task); intra-grain calls (interactions between parallel objects in the same grain) are serialised and synchronous. Figure 2a shows an example with three non-packed parallel objects. In Figure 2b, parallel objects 1 and 2 are packed into a single grain and calls from object 1 to object 2 are synchronous, whereas calls from object 1 to object 3 (inter-grain calls) may still be asynchronous. Intra-grain calls (as method calls from object 1 to object 2) may be performed directly (through direct procedure call) instead of a more costly call though the IPC. The creation and the destruction of a parallel object inside a grain (such as object 2) may also be a direct object creation/destruction instead of a parallel object creation/destruction through the run-time system.

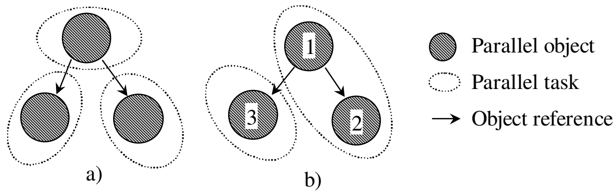

Figure 2. Parallel objects without a) and with b) grain packing

Cost benefits from the grain packing approach are traded with performance, since the decision to pack takes time and packed objects may have a greater latency when replying to external requests, due to thread/process sharing among objects in the same grain.

Since parallel objects may perform intermediate communication, i.e. they may exchange messages after start executing a method, cyclic communication may be introduced by the grain packing mechanism. On intra-grain direct calls, swapping an asynchronous method call to synchronous may originate deadlock, since all objects in a grain are bind to a single thread/process. It is avoided by tracing intra-grain calls and detecting cyclic calls. A call closing a cycle is not performed synchronously/directly, but by a message requesting the method execution. A similar problem may occur on synchronous inter-grains calls, which is prevented by spanning a new thread to manage these calls.

Dynamic grain control is further enriched with the capability to reverse packing decisions, by splitting a packed grain into its original parallel tasks.

The decision “when to adapt” is taken according to run-time evaluation of the system load. In the current SCOOPP prototype the grain packing should occur when the system is highly loaded and grain unpacking should occur when the system is lightly loaded. To evaluate the overall system load several techniques are under study. These include run-time measurements of task queue length, processor computational load, inter-process communication load and a mix of the computing and network load.

To implement the grain control policy, both for grain packing and grain unpacking, tasks are selected according to the interrelationship between parallel objects. One approach is to pack objects randomly, which may reduce the task management overhead; however, the inter-object communication overhead can be reduced if related parallel objects are packed. Objects with stronger inter-task dependence, i.e., with many cross calls or large data exchanged between them, are the best candidates for task merging in a single grain.

Experimental results from related work [14] also suggest the advantage of placing several grains per processor, but further experiments are still required to achieve optimal values.

5. The ParC++ prototype

The current SCOOPP prototype, ParC++ [1], supports some extensions to C++ and consists of a ParC++ to C++ pre-processor, several support classes and a run-time system. This prototype run on top of several programming environments, namely the INMOS toolset and Parix. Support classes for PVM are under tests.

Current ParC++ run-time system creates on each node a pool of local processes. Whenever a method request arrives to a node, a free process is scheduled to serve it, executing the requested method on the destination object. On each node a special parallel object performs load balancing and manages the pool of processes. It permanently tunes the number of processes per node, according to the current policy to perform the grain packing/unpacking. The current prototype defines a limit on the maximum number of concurrent processes per processor at compile-time; this limit represents the maximum number of grains per node; other strategies are under study, including dynamic ones.

At system start-up, a pool of processes is created at each node. As parallel objects are created and spread across the system (following a load distribution policy), the processes in the pool serve requests for the created objects, until the number of parallel objects reaches the predefined limit of processes per processor. Once there, newly created parallel objects are grain packed by sharing processes, holding the number of processes. Later, when the overall number of parallel objects decreases, objects may be grain unpacked and migrate to remote processing nodes, according to the load policy.

Grain packed objects may perform both direct method call and direct object creation/destruction on local objects (intra-grain operations). Direct method calls are performed if an inter-object cyclic call is not detected; otherwise the call is asynchronously performed through the IPC. The run-time system is notified of all direct creation/ destruction of objects, to support later grain unpacking by the run-time system.

6. Performance results

The SCOOPP approach on the ParC++ prototype is evaluated measuring the execution times of a ParC++ program on two distributed memory parallel systems: a PowerExplorer (16 nodes, each with a PowerPC as a computing processor and a Transputer as a communication processor) and a MC-3 (with 112 Transputer based nodes), running Parix versions 1.3 and 1.2, both providing a high clock precision of 1µs. The measurements were taken on the ParC++ system with support classes version 1.13 and the pre-processor version 1.61.

The ParC++ application has an hierarchical farming structure; its algorithm is often used in parallel applications. It includes two main types of parallel activities (i.e., parallel objects): the farmer process to split the work into several frames and to distribute them, and the worker process, to execute the frames and to return the results. When the farmer receives a processed frame, it merges it with the processed work and sends a new frame if there are more frames to process. If there are no more frames to process the farmer sends a termination order to the worker.

The ParC++ farming application works in a 4-level hierarchical scheme: each farmer has 7 workers. Each worker acts like a farmer and has also 7 workers. The lowest level has 343 workers (from a total of 400 objects) which do the work, while all the other levels just split the work and merge the processed work. Since farming works on a demand driven scheme - faster workers will process more frames - the application is well balanced, reducing the impact of the load balancing overhead.

Without grain-size adaptation (i.e., with fixed grains) all parallel objects are implemented as parallel tasks, which is the finest grain-size. By merging at run-time several parallel objects into a single parallel task, forming a larger grain, SCOOPP is dynamically adapting the grain-size.

Execution times were measured for a local 11x11 threshold on an 256x256 image recursively divided into 16 frames, using fixed grain-size and enlarged grain-size. The results are presented in Table 1 and 2. Parallel objects where created using a random load distribution policy and establishing a limit of 3 grains per node for grain-size adaptation.

MC-3			
Processors	Fixed grains	Enlarged grains	Gain
4	41.90	34.72	17%
7	23.80	18.70	21%
14	9.98	8.78	12%
28	4.83	4.60	5%
56	2.72	2.73	0%
112	1.80	1.83	-2%

Table 1 - Execution times (in seconds) on a MC-3.

On the MC-3 and when the number of object per processor is high (i.e., under a low number of processors) the grain-size adaptation performed considerable better than a fixed grain-size strategy, obtaining a maximum reduction in the execution time of 21% in 7 processors.

When the average number of objects per processor gets closer to the unit (in this test occurs when the number of processors is more than 56) the overhead penalty to pack grains overpasses the benefits of grain packing.

Execution times on the PowerExplorer (Table 2) reinforce the effectiveness of the grain-size adaptation strategy of the SCOOPP system. Running the previous program without changes on this platform provides a reduction on execution times up to 44%, due to the excess of parallelism and to the higher ratio computation/ communication of this machine, providing more room for performance increase through grain packing.

PowerXplorer			
Processors	Fixed grains	Enlarged grains	Gain
4	5.50	3.07	44%
8	2.55	1.73	32%
12	1.73	1.27	27%
16	1.36	1.06	22%

Table 2 - Execution times (in seconds) on a PowerXplorer.

These results show that under excess of parallelism the grain-size adaptation performs well, but under a small number of parallel activities per node (near the limit of the algorithm scalability) a small performance penalty occurs due to the system attempt to perform grain packing. To be effective on these conditions the system should further unpack the programmer specified parallel grains, using additional parallelism extraction features.

7. Conclusions and future work

The SCOOPP system frees the programmer to express the full potential parallelism of an algorithm, in a platform independent way. It performs the grain-size adaptation at run-time to obtain the best granularity for the target platform, improving the algorithm scalability over multiple platforms without programmer intervention.

Conventional grain control approaches usually limit the grain packing within independent tasks with fork/join synchronisation. The proposed methodology in SCOOPP to dynamically adapt the grain-size also supports any kind of tasks, including non related ones and those which exchange messages during its execution.

The results obtained on the current prototype showed the effectiveness of the automatic grain-size adaptation policy on applications with excess of parallelism. On such cases, grain packed objects have a considerable lower overhead when compared with non-packed objects, improving the algorithm performance without user intervention. However, when an algorithm lies near the limit of its scalability (having few parallel task per node) the performance penalty due to the attempt to perform grain packing can be reduced through parallelism extraction.

The results obtained so far are very encouraging, but further work is still needed to improve the system performance through granularity control; this includes:

- techniques to evaluate the overall system load at run-time, including both the computational and the communication loads, and their ratio;
- methodologies to evaluate, at run-time, the number of grains per processor which defines the crossover value on performance;
- performance improvements through the use of qualitative grain-size adaptation, i.e., how to select the most adequate objects to be grain packed/unpacked;
- further refinements on the run-time decision process to unpack, namely support for object migration.

Current work also includes further improvements to the run-time system based on practical applications and the porting of the prototype system to other programming environments such as PVM and MPI.

These SCOOPP improvements will be evaluated through deeper and more comprehensive parallel applications, taken from non-academic case studies.

References

[1] J.Sobral, A.Proença. *ParC++: A Simple Extension of C++ to Parallel Systems*, Proc. 6th Euromicro Workshop Parallel and Distributed Processing, Madrid, Spain, January 1998.

[2] B.Kruatrachue, T.Lewis. *Grain Size Determination for Parallel Processing*, IEEE Software, v5(1), January, 1988.

[3] A.Gresoulis, T.Yang. *On the Granularity and Clustering of Direct Acyclic Graphs*, IEEE Trans. on Parallel and Distributed Systems, v4(6), June 1993.

[4] B.Andersen. A General, *Fine-Grained, Machine Independent, Object-Oriented Language*, ACM SIGPLAN Notices, v29(5), May 1994.

[5] J.Larus, B.Richards, G.Viswanathan, *C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language*, in Parallel Programming Using C++, The MIT Press, 1996.

[6] P.Beckman, D.Gannon, E.Johnson. *HPC++ and the HPC++ Lib. Toolkit*, White Paper, www.extreme.indiana.edu/hpc++, 1997.

[7] C.Chang, A.Sussman, J.Saltz *CHAOS++*, in Parallel Programming Using C++, The MIT Press, 1996.

[8] E.Mohr, A.Kranz, R.Halstead. *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*, IEEE Trans. on Parallel and Distributed Processing, v2(3), July 1991.

[9] P.Lopez, M.Hermenegildo, S.Debray. *A Methodology for Granularity Based Control of Parallelism in Logic Programs*, Journal of Symbolic Computation, v22, 1998.

[10] L.Kale, S.Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*, OOPSLA '93, ACM SIGPLAN Notices, v28(10), October 1993.

[11] J.Poole. *UC++ 2: User Manual*, (draft), The London Parallel Applications Center (LPAC), June 1998.

[12] R.Chandra, A.Gupta, J.Hennessy. *COOL: An Object-Based Language for Parallel Programming*. IEEE Computer, v27(9), August 1994

[13] A.Grimshaw. *Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing*, ACM Transactions on Computer Systems, v14(2), May 1996.

[14] A.Chalmers, A.Proença, *A messages density monitoring strategy for distributed memory parallel system*, Proc. 2nd Int. Conf. on Software for Multiprocessors and Supercomputers, Moscow, September 1994.