A Run-time System for Dynamic Grain Packing

João Luís Sobral, Alberto José Proença

Departamento de Informática - Universidade do Minho
4709 Braga Codex - PORTUGAL
{jls, aproenca}@di.uminho.pt

Abstract. The SCOOPP (Scalable Object Oriented Parallel Programming) system is an hybrid compile and run-time system, that extracts parallelism, supports explicit parallelism and dynamically serialises parallel tasks in excess, to dynamically scale applications through a wide range of target platforms.
This paper describes the run-time system of the current SCOOPP prototype - the ParC++ - and its mechanism to serialise parallelism. Low level performance results are presented, which indicate that the proposed methodology is effective and provides an high reduction in parallelism overheads. These features can improve the scalability of parallel applications with excessive parallelism.

1 Introduction

When developing parallel applications, programmers are often faced with a parallelism granularity control decision: a larger number of fine parallel grains may help to scale up the parallel application and it may also improve the load balancing; however, it requires more control overhead, and performance may degrade due to excessive communication over computation, if grains are too fine.

Static granularity control [1][2] is usually applied to fine grained tasks, whose behaviour is known at compile-time. However, parallel applications with irregular parallel tasks - where tasks behaviour is not known at compile-time - require dynamic granularity control to achieve an acceptable performance. Programmer based dynamic granularity control adds an extra burden on the programmer activity, requires a deep knowledge of both architecture and algorithm behaviours and decreases the program clarity.

The SCOOPP system [3] is an hybrid compile and run-time system, that extracts parallelism, supports explicit parallelism and dynamically serialises parallel tasks in excess at run-time to dynamically scale applications through a wide range of target platforms. This paper focus on the methodology to remove excess parallelism in the

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).

Euro-Par'99, LNCS 1685, pp. 845–852, 1999.

run-time system in the current SCOOPP prototype (ParC++) and it attempts to measure its impact on performance.

Section 2 presents the SCOOPP system and Section 3 discusses the alternatives to dynamically remove excess parallelism. Section 4 and 5 present the ParC++ system and some performance results. Section 6 concludes the paper with suggestions for future work.

2 SCOOPP System Overview

Explicit parallelism is specified in SCOOPP through a special type of object: the parallel object. These objects model parallel activities and may be placed at remote processing nodes. Parallel objects communicate through methods calls, either asynchronously, when no return value is expected (i.e. the caller does not wait for method completion) or synchronously, when a value is expected. References to parallel objects may be freely copied or used as method arguments, supporting the development of parallel applications with complex inter-object communication graphs. When this feature is not used the inter-object communication graph is a tree.

Parallel objects may also create “sequential objects”, taking advantage of existing code. These objects are placed in the context of the parallel object which created them and only copies of them are allowed to move between parallel objects. Method calls on these objects are always synchronous.

The SCOOPP system granularity control is accomplished in two steps: at compile-time - the compiler and/or the programmer specifies a large number of parallel objects - and at run-time - parallel objects are packed into larger grains, according to the application/target platform behaviour and based on security and performance issues.

Parallelism extraction is performed by transforming selected sequential objects into parallel objects, whereas parallelism serialisation (i.e. grain packing) is performed by transforming parallel objects into sequential ones. In this serialisation process, compiler transformed parallel objects are preferred for serialisation; parallel objects are only serialised when all compiler transformed parallel objects have been serialised (more details in [6]).

3 Removing Excess Parallelism

Conventional grain packing mechanisms [4][5] are based on fork/join parallelism. Grain-size is increased by ignoring the fork construct and executing tasks sequentially, instead of spanning a new parallel activity to execute the forked task. This mechanism to increase the grain-size is fault free when tasks have no intermediate communication, i.e., inter-tasks dependencies form direct acyclic graphs. However, parallel objects (tasks) may perform intermediate communications, which, in turn, may generate rather complex inter-tasks communication graphs,

namely, on explicit parallelism. Alternative mechanisms are required to increase the grain-size and these should guarantee a correct program behaviour.

The SCOOPP system packs grains by joining several parallel objects in a single computing grain and by serialising intra-grain operations. Intra-grain method calls - between objects within the same grain - are synchronous and are usually performed directly as a normal procedure call. Asynchronous inter-grain calls are implemented through standard inter-tasks communication mechanisms.

On the SCOOPP ParC++ prototype, the development of the grain packing mechanism was guided by the following requirements:

- *correctness* - grain packed tasks should have the correct behaviour, namely, packing should not introduce deadlock;
- *reversibility* - the system should provide both packing and unpacking operations;
- *fairness* - packed tasks should have the same opportunities of execution as non packed tasks.

To ensure program *correctness* some intra-grain operations should not be serialised. Deadlock may occur with cyclic inter-grain communication - when asynchronous method calls are changed to synchronous (Fig.1a) - and in inter-grain synchronous calls when packing cross referencing objects in the same grain (Fig.1b); in this case, one of the grains must span a new thread to serve the incoming request, to avoid a deadlock.

Without grain packing, both calls in Fig.1a are executed simultaneously by two separate threads/processes. Since calls are performed asynchronously, if either object is busy, the request is queued into the object message buffer. When these grains are packed together and calls are changed to synchronous, the same thread must execute both calls. It may first execute the call from object a to object b suspending the execution of a and executing object b code. Later, when executing a call from object b to object a, deadlock will occur if object a can not be re-entered. To avoid this, each thread traces the intra-grain calls and when a cyclic call is detected the call closing the cycle is not performed directly, but through a standard inter-object communication mechanism.

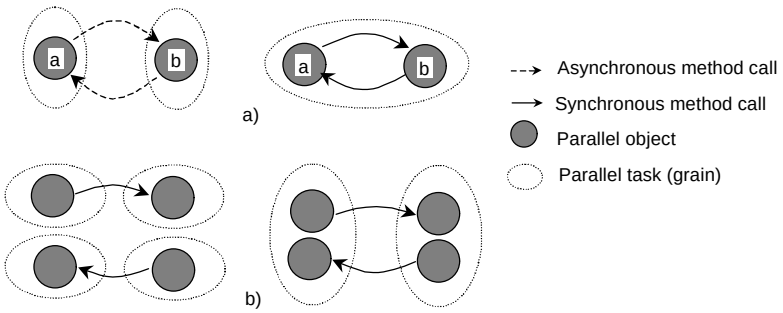


Fig. 1. Possible deadlock situations for grain packing: a) in intra-grain cyclic calls; b) in inter-grain synchronous calls

To support the *reversibility* of the packing process, packed parallel objects are not transformed into "pure" sequential objects. Instead, they remain with "parallel object" functionality, but they share a thread with other objects in the same grain. As a consequence, the management of packed objects has a small overhead when compared with sequential objects. However, reversing the packing process is simpler and faster on this approach, since packing/unpacking is just sharing/not sharing a thread among parallel objects. Moreover, remote access to packed objects is simplified since it is an intrinsic feature of parallel objects.

To support *fairness*, requests for method calls are not allowed to wait longer than a predefined time slot before being executed, reducing the deadlock impact showed in Fig.1b. If *s* is the time-slice allocated to each thread and *n* the number of threads on a node, any request to a free object will start being served in less than *(n-1)*s*, without grain packing. With grain packing, a longer wait may occur, since packed parallel objects may suffer starvation from a long running method call in the same grain, as a consequence of the intra-grain serialisation. The implemented maximum waiting for external requests is directly proportional to the number of objects on each node.

Fairness is further improved by establishing a non-fixed association among threads and computing grains, i.e. any local thread may serve a request for any local object (Fig.2). In contrast to a fixed association model, this non-fixed association gives equals execution opportunities to all the objects in a processing node; it automatically balances the size of each grain, and it automatically performs local packing/unpacking operations, since threads are associated to objects in requests arrive. However, it does require additional checks on method calls for inter-thread synchronisation.

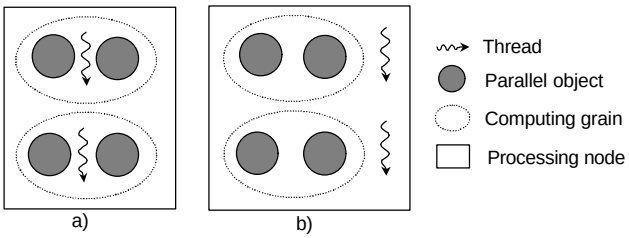


Fig. 2. Fixed a) and non-fixed b) association between threads and computing grains

4 The Run-time System in ParC++ Prototype

The current SCOOPP ParC++ prototype, supports some extensions to C++. It includes a ParC++ pre-processor, several C++ support classes and a run-time system. The pre-processor analyses the application - by retrieving information about the declared parallel objects - and generates code for remote object creation and remote method invocation. It may also mark C++ objects to be transformed into parallel

objects by the run-time system. The prototype has been tested on several distributed memory parallel architectures: a Parsytec MC-3 (with 112 Transputer based nodes), a Parsytec PowerExplorer (16 nodes, each one with one PowerPC as a computing processor and one Transputer as a communication processor) and PC clusters.

The ParC++ run-time system (RTS) is based on three object classes: proxy objects (PO), implementation objects (IO) and server objects (SO).

A PO represents a local or a remote parallel object and has the same interface as the object it represents. It transparently replaces a parallel object and forwards all method invocations to the IO, the object that implements the parallel object methods. SO are active entities (i.e. threads) that continuously receive messages from PO objects, calling the requested method on local IO and, if needed, returning the result value to the caller. A PO maintains the node address of the IO, as well as its system identification and the identification of its SO (which is shared by all SO for packed objects, under the non-fixed association model).

On inter-grains method calls the PO forwards the call to a remote SO, which activates the corresponding method on the IO (calls a in Fig.3). On intra-grain calls, the PO directly calls the corresponding method on the local IO (call b in Fig.3), since the PO is placed on the same processing node as the IO. To preserve the program correctness, when the IO is busy - when it is executing a method call that does not allow another concurrent method execution or under a cyclic call - the call is forwarded to a local SO, which will delay the request until the IO is ready.

The RTS provides run-time grain-size adaptation and load balancing through the object manager (OM). The application entry code creates one instance of this object on each processing node. The OM controls the grain-size adaptation by determining the number of SO's per node. This decision is based on run-time measurements of traffic and computing loads, currently undergoing work on related I&D projects [7].

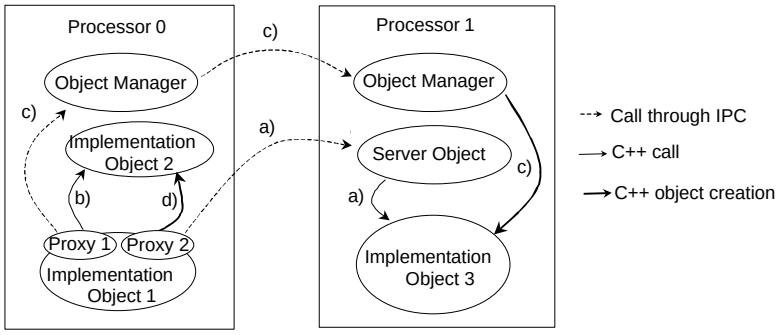


Fig. 3. Inter-grains a) and intra-grain b) method calls; RTS c) and d) direct object creation

On receiving a request to create a parallel object, the RTS creates a new local PO. The first task of the newly created PO is to request the creation of the IO. When parallelism is not being removed, the OM selects a processing node to create a new

IO (according to the current load distribution policy), selects or creates the associated SO and returns their identifier to the PO (calls c in Fig.3). When the RTS is removing excess parallelism, the PO directly creates the parallel object, by locally creating the IO (call d in Fig.3) and notifying the RTS. The PO always destroys a local IO; non local objects are destroyed by the RTS, upon a request from the PO.

The RTS may temporarily increase the number of SO's in a node to prevent deadlock and improve fairness. This happens when external requests are waiting longer than a predefined time. Additional SO's may be destroyed when the requests are completed.

5 Performance Evaluation

One of the main goals of SCOOPP is dynamic scalability. To measure its impact on performance, a low level evaluation was performed. The evaluation runs measures the execution times of different size tasks (related to granularity) and the overhead time to manage the parallel objects on 2 parallel systems. These benchmarks systems had no application objects, and both used point-to-point Transputer based communications: a PowerExplorer with Parix 1.3 and a MC-3 with Parix 1.2 (both providing a clock precision of 1μs). The measured values were taken on the ParC++ system with support classes version 1.14 and the pre-processor version 1.61.

The main overheads in OO applications include the object creation, the method calls and the object destruction. These operations usually correspond to object space allocation and initialization, procedure calls and object space de-allocation. On parallel OO applications, the additional overheads to manage parallel objects in SCOOPP include the RTS creation of PO, SO and IO (on object creation), the proxy request for remote method execution (on method calls) and the RTS deletion of PO, SO and IO (on object destruction).

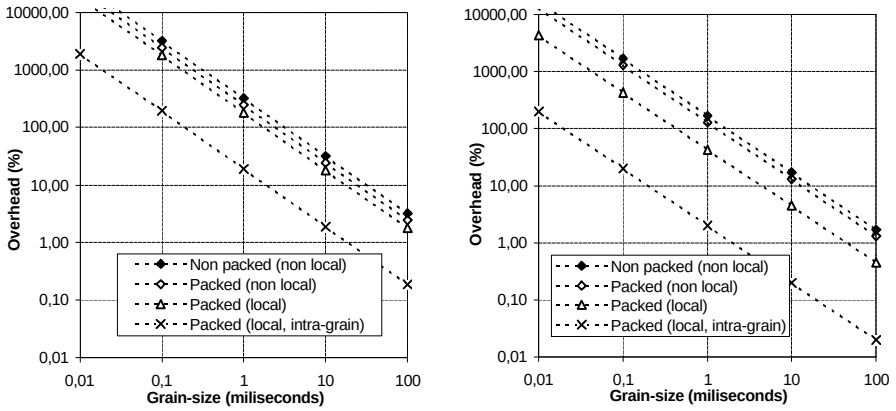
The evaluation runs aim to measure the ratio between the objects management overhead and the execution time of the object's computational task (the object grain-size). Fig.4 shows 4 sets of measured times, required to create, activate and destroy one parallel object on a 2 node system:

- with no granularity control (non packed objects), with remote placement;
- with granularity control (packed objects) and remotely packed;
- packed into another local grain;
- included in the source grain.

The first set has no RTS optimisation, while the other three show different approaches to remove excess of parallelism.

An overhead of 100% in Fig.4 means that the time required to manage a parallel object equals the effective execution time of the required task, i.e., it is an indication of a crossover value to make the decision to pack objects, under these benchmark conditions. When adding more computing nodes or computational tasks, all the curves move up in Fig.4.

For an overhead of 100%, Fig.4 shows that remotely packed objects improves the overhead ratio by supporting efficient finer parallelism granularity, mainly due to object creation time: the server object creation (and its process) is not required for non-local packed objects. This improvement is strongly dependent on the process management overheads (process creation, scheduling and destruction).



a) 30 MHz T805 (in MC-3) b) 60 MHz PPC 601 (in PowerXplorer)

Fig. 4. Parallelism overhead as a function of the grain-size

Further improvement is obtained when these packed objects are also locally placed, due to a swap from non-local to local IPC communication, and depends on communication latency and bandwidth.

A significant improvement can be seen when parallel objects are packed into the source grain, since the RTS operations on packed objects are replaced by the proxy intervention, where IPC is swapped by a procedure call.

When the evaluation runs moves from MC-3 to the PowerXplorer, the improvements in parallelism granularity – i.e., the finest supported grain-size for a given overhead ratio – are from one to two orders of magnitude, only due to a faster processor at the computing nodes, with the same communication processor and bandwidth. This suggests that with faster processors, the overhead ratio is further improved. However, if faster communication lines are used, it can be expected no change on local packing and a significant improvement on remotely packed objects.

Other low level results have been presented in [8] and application oriented performance results have been presented in [3], which obtained a reduction on an image processing application's execution time down to 79% on the MC-3 and 56% on the PowerXplorer.

Conclusions and Future work

A dynamic grain packing mechanism largely decreases the minimum cost of an programmer/pre-processor specified parallel task, which allows the expression of parallel OO applications in a natural way. Grain packing should encourage the programmer and/or the pre-processor to detect and specify all the parallelism opportunities and let the RTS to remove the excess parallelism.

Evaluation of the current ParC++ prototype showed that the specification of parallel objects through a new keyword – which required a pre-processor – also allows a natural implementation of implicit parallelism extraction by the same pre-processor. This feature, together with the use of proxy concept to implement remote parallel objects, provides a powerful tool for a transparent dynamic (run-time) grain-size adaptation. The results obtained so far show that serialising parallel activities (by increasing the grain-size) provides an high reduction in the parallelism overheads.

Further work is currently being performed to complete the dynamic grain packing mechanism. It includes the development of techniques to determine: (i) when to perform the grain packing/unpacking based on the system load, (ii) which objects should be packed/unpacked, and (iii) on which grains.

Current R&D work also includes further evaluation of the prototype with case studies from virtual environment generation.

References

1. Kruatrachue, B., Lewis, T.: Grain Size Determination for Parallel Processing, IEEE Software, Vol. 5(1), January (1988)
2. Gresoulis, A., Yang., T.: On the Granularity and Clustering of Direct Acyclic Graphs, IEEE Transactions on Parallel and Distributed Systems, Vol. 4(6), June (1993)
3. Sobral, J., Proença., A.: Dynamic Grain-Size Adaptation on Object-Oriented Parallel Programming - The SCOOPP Approach, Proc. of the 2nd Merged IPPS/SPDP 1999, Puerto Rico, April (1999)
4. Mohr, E., Kranz, A., Halstead, R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, IEEE Transactions on Parallel and Distributed Processing, v2(3), July (1991)
5. Lopez, P., Hermenegildo, M., Debray, S.: A Methodology for Granularity Based Control of Parallelism in Logic Programs, Journal of Symbolic Computation, Vol. 22, (1998)
6. Sobral, J., Proença, A.: ParC++: A Simple Extension of C++ to Parallel Systems, Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Applications (PDP'98), Madrid, Spain, January (1998)
7. Santos, L., Chalmers, A., Proença, A.: A messages density monitoring strategy for distributed memory parallel system, 2nd Int. Conf. on Software for Multiprocessors and Supercomputers: theory, practice and experience, Moscow, September (1994)
8. Sobral, J., Proença, A.: Overheads on the dynamical removal of excess of parallelism on OO irregular applications, 1st Work. Parallel Computing for Irregular Appl., 5th Int. Symp. HPC Arch.(HPCA-5), www-apache.imag.fr/manifestations/PCIA, Orlando, January (1999)