A SCOOPP Evaluation on Packing Parallel Objects in Run-time*

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 is an hybrid compile and run-time system. SCOOPP dynamically scales OO applications on a wide range of target platforms, including a novel feature to perform a run-time packing of excess parallel tasks. This communication details the methodology and policies to pack parallel objects into grains and method calls into messages. The SCOOPP evaluation focus on a pipelined parallel algorithm - the Eratosthenes sieve - which may dynamically generate a large number of fine-grained parallel tasks and messages. This case study shows how the parallelism grain-size - both computational and communication - has a strong impact on performance and on the programmer burden. The presented performance results show that the SCOOPP methodology is feasible and the proposed policies achieve efficient portability results across several target platforms.

1 Introduction

Most parallel applications require parallelism granularity decisions: a larger number of fine parallel tasks may help to scale up the parallel application and it may improve the load balancing. However, if parallel tasks are too fine, performance may degrade due to parallelism overheads, both at the computational and communication level.

Static granularity control, performed at compile-time, can be efficiently applied to fine grained tasks [1][2], whose number and behaviour is known at compile-time. HPF [3], HPC++ [4], and Ellie [5] are examples of environments that support static granularity control. However, parallel applications where parallel tasks are dynamically created and whose granularity can not be accurately estimated at compile-time require dynamic granularity control to get an acceptable performance; this also applies when portability is required across several platforms.

Granularity control can lead to better performance when performed by the programmer, but it adds an extra burden on the programmer activity: it requires knowledge of both the architecture and the algorithm behaviour, and it also reduces the code clarity, reusability and portability.

* 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).

The SCOOPP system [6] 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 evaluates the application of the SCOOPP methodology to dynamically scale a pipelined application - the Eratosthenes sieve - on three different generations of parallel systems: a 7 node Pentium II 350MHz based cluster, running Linux with a threaded PVM on TCP/IP, a 16 node PowerPC 601 66 MHz based Parsytec PowerXplorer and a 56 node T805 30Mhz based Parsytec MultiCluster 3, both running PARIX with proprietary communication primitives, functionally identical to PVM. The cluster nodes are inter-connected through a 1 GBbit Myrinet switch, the PowerXplorer nodes use a 4x4 mesh of 10Mbit Transputer-based connections and the MultiCluster Transputers are interconnected through a 7x8 mesh.

Section 2 presents an overview of the SCOOPP system and its features to dynamically evaluate the parallelism granularity and to remove excess parallelism. Section 3 introduces the Eratosthenes sieve and presents the performance results. Section 4 concludes the paper and presents suggestions for future work.

2 SCOOPP System Overview

SCOOPP is based on an object oriented programming paradigm supporting both active and passive objects. Active objects are called parallel objects in SCOOPP (//obj) and they specify explicit parallelism. These objects model parallel tasks and may be placed at remote processing nodes. They communicate through either asynchronous or synchronous method calls.

Passive objects are supported to take advantage of existing code. These objects are placed in the context of the parallel object that created them, and only copies of them are allowed to move between parallel objects. Method calls on these objects are always synchronous.

Parallelism extraction is performed by transforming selected passive objects into parallel objects (more details in [7]), whereas parallelism serialisation (i.e. grain packing) is performed by transforming parallel objects into passive ones [8].

Granularity control in SCOOPP is accomplished in two steps. At compile-time the compiler and/or the programmer specifies a large number of fine-grained parallel objects. At run-time parallel objects are packed into larger grains - according to the application/target platform behaviour and based on security and performance issues – and method calls are packed into larger messages.

Packing methodologies are concerned on “how” to pack and “which” items to pack; this subject is analysed in section 2.1. These methodologies rely on parameters, which are estimated to control granularity at run-time; these are analysed on section 2.2. Packing policies focus on “when” and “how much” to pack, and they heavily rely on the structure of the application; this subject is analysed in section 2.3.

2.1 Run-time Granularity Control

Conventional approaches for run-time granularity control are based on fork/join parallelism [9][10][11][12][13]. The grain-size can be increased by ignoring the fork and executing tasks sequentially, avoiding spawning a new parallel activity to execute the forked task.

The SCOOPP system dynamically controls granularity by packing several //obj into a single grain and serialising intra-grain operations. Additionally, SCOOPP can reduce inter-grains communication by packing several method calls into a single message.

Packing Parallel Objects. The main goal of object packing is to decrease parallelism overheads by increasing the number of intra-grain operations between remote method calls. Intra-grain method calls - between objects within the same grain - are synchronous and usually performed directly as a normal procedure call; asynchronous inter-grain calls are implemented through standard inter-tasks communication mechanisms.

The SCOOPP run-time system packs objects when the grain-size is too fine and/or when the system load is high. The SCOOPP system takes advantage of the availability of granularity information on existing //obj. When parallel tasks (e.g. //obj) are created at run-time, it uses this information to decide if a newly created //obj should be used to enlarge an existing grain (e.g. locally packed) or originate a new remote grain.

Packing Method Calls. Method call packing in SCOOPP aims to reduce parallelism overheads by packing several method calls into a single message.

The SCOOPP run-time system packs method calls when the grain-size is too fine. On each inter-grains method call, SCOOPP uses granularity information on existing objects to decide if the call generates a new message or if it is packed together with other method calls into a single message.

Packing Parallel Objects and Method Calls. The two types of packing complement each other to increase the grain-size. They differ in two aspects: (i) method calls can not be packed on all applications, since the packing relies on repeated method calls between two grains, and may lead to deadlock when calls are delayed for an arbitrary long time; this delay arises from the need to fulfil the required number of calls per message; (ii) method calls in a message can be more easily unpacked than objects in a grain. Reversing object packing usually requires object migration, whereas packs of method calls can be sent without waiting for the message to be fully packed. In SCOOPP, packs of methods calls are sent either on programmer request or when the source grain calls a different method on the same remote grain.

2.2 Parameters Estimation

To take the decision to pack, two sets of parameters are considered: those that are application independent and those that are application dependent. The former includes the latency of a remote "null-method" call (α) and the inter-node communication bandwidth. The later includes the average overhead of the method parameters passing (ν), the average local method execution time (μ), the method fan-out (ϕ) (e.g., the average number of method calls performed on each object per method execution) and the number of grains per node (γ).

Application independent parameters are statically evaluated by a kernel application, running prior to the application execution; parameters that depend on the application are dynamically evaluated during application execution. The next two subsections present more details of how these two types of parameters are estimated.

Application Independent Parameters. Application independent parameters include the latency of a remote "null-method" call (α) and the inter-node communication bandwidth. Both parameters are defined for a “unloaded” target platform. They are estimated through a simple kernel SCOOPP application that creates several //obj on remote nodes and performs a method call on each object.

The remote method call latency (α) is the time required to activate a method call on a remote //obj. It is estimated as half the time required to call and remotely execute a method that has no parameters and only returns a value.

The inter-node communication bandwidth is estimated by measuring the time required to call a method with an arbitrary large parameters size. It is half of the division of the parameters size by the time required to execute the method call.

On some target platforms, these two parameters depend on the pair source/destination nodes, namely on the interconnection topology. In such cases, the SCOOPP computes the average from the parameters taken between all pairs of nodes. Moreover, these parameters tend to increase when the target platform is highly loaded, due to network congestion and computational load. However, this effect is taken into account on the SCOOPP methodology through the γ parameter (number of grains per node), which is a measure of the load on each node.

These two parameters are statically estimated to reduce congestion penalties at run-time, since they require inter-node communication, which is one of the main sources of parallelism overheads. Their evaluation at run-time, during application execution, may introduce a significant performance penalty.

Application Dependent Parameters. SCOOPP monitors granularity by computing, at run-time, the average overhead of the method parameters passing (ν), the average method execution time (μ) and the average method fan-out (ϕ). SCOOPP computes these parameters, at each object creation, from application data collected during run-time.

The overhead of the method parameters passing (ν) is computed from the inter-node communication bandwidth multiplied by the average method parameter size. This last one is evaluated by recording the number of method calls and adding the parameter sizes of each method call.

The average method execution time (μ) is evaluated by recording the time required to perform each local method execution. When a method does not perform other calls, this value is just the elapsed time. When a method contains other calls, the measurement is split into the pre-call and after-call phases, and the previous procedure is applied to each phase. Moreover, the time required to perform the pre-call phase is used has a first estimate of the average method execution time, so that the average method execution time data is available for the next method call, even if the first method execution one has not completed yet.

The average method fan-out (ϕ) is measured by a global program analysis through object and method calls statistics. The run-time system marks each //obj with its depth on the object creation tree. The depth of the root object is one and the depth of all other //obj is equal to the depth of its creator plus one. The run-time system maintains a table for the number of call performed on each depth, which is incremented on each local method call. The method fan-out is derived from this table through the overall ratio between consecutive depths.

SCOOPP minimises the run-time impact of the parameters estimation overhead in three ways. First, granularity information is collected at class level, e.g., the ν, μ and ϕ parameters are measured for each class of parallel objects. This approach is clearly less costly than an instance-based approach and more accurate than a global one. Second, when the overhead introduced to access the system clock to measure the average method execution time is high (usually more than 1%) the frequency of information retrieval is reduced; this excludes, however, the application start up phase, since on that phase no information is available. Third, the parameters that are estimated at run-time do not require inter-nodes communication, since the estimation is locally performed and parameters information is only exchanged within requests for remote object creation.

2.3 Packing Policies

Packing policies define “when” and “how much” to pack, e.g. the number of //obj that should be packed in each grain, and the number of method calls to pack on each message. These policies are usually grouped according to the structure of the application: object pipelines, static object trees (e.g. object farming) and dynamic object trees (e.g. work split and merge). The work here presented focus on packing policies for pipelined algorithms and the next section evaluates its application to a case study, the Eratosthenes sieve.

Packing Parallel Objects. The decision “when” to pack is taken based on the average method execution time (μ), the average latency of a remote "null-method" call (α) and the overhead of the method parameters passing (ν). When the average method execution time is excessively short, //obj should be packed, which occurs when the overhead of a remote method call is higher than the average method execution time, e.g., $(\alpha+\nu)>\mu$. This is the turnover point to pack //obj, where the parallelism overhead becomes longer than the time spent on locally "useful work".

The decision of "how many" //obj to pack into a single grain (e.g., degree of object packing or computation grain-size, C_p) is related to the α, ν and μ parameters as seen before, and also on the method fan-out (ϕ) and on the system computational load, e.g., the number of grains per node (γ). The computation grain-size should be increased when the system presents high parallelism overhead (e.g., high α and ν) and be decreased on high average method execution time. The degree of object packing should also be decreased when fan-out increases, since each method call performs several intra-grain calls, and it can be increased when the number of grains per node is high, to decrease parallelism overheads.

On pipelined applications, packing adjacent //obj makes the number of intra-grain calls equal to the average number of objects in each grain, since the fan-out is close to 1. When C_p //obj are packed together, each remote method call generates C_p method calls, executing on $C_p\mu$ time. Under these conditions, the turnover point to decide when to pack is reached when $C_p =(\alpha+\nu)/\mu$. This expression defines the minimum number of //obj to pack on each grain to overcome the parallelism overheads. To decrease parallelism overheads even more, SCOOPP increases the number of //obj on each grain linearly with γ by using the expression $C_p=\gamma(\alpha+\nu)/\mu$.

Packing Method Calls. The decision "when" to pack method calls follows the same rule as the one applied to pack objects, e.g., when $(\alpha+\nu)>\mu$; this condition reflects that the overhead to place a single remote call is higher than the remote method execution time. In this case, several inter-grains calls should be packed to reduce communication overheads.

The decision "how many" method calls to pack into a single message (e.g., degree of method call packing or communication grain-size, C_m) is computed from the α, ν and μ parameters. Sending a message that packs C_m method calls has a time overhead of $(\alpha+C_m\nu)$ and the time to locally execute this pack is $C_m\mu$. Packing should be performed such that $(\alpha+C_m\nu)<C_m\mu$, e.g., when the overhead to place a remote call is lower than the time to locally execute the pack of method calls. Resolving the equation gives the turnover point $C_m=\alpha/(\mu-\nu)$.

When the average method execution time is close or smaller than the overhead of the parameter passing (e.g., $\mu<=\nu$), method calls should not be packed. However, this rule can be relaxed if both method calls and //obj are packed.

Packing Parallel Objects and Method Calls. SCOOPP can simultaneously pack method calls and //obj. However, when method calls are packed, the application performance may benefit from a less //obj packing degree. In this case, SCOOPP scales down the computation grain-size by using the expression $C_p=\gamma(\alpha+C_m\nu)/(\mu C_m)$.

When the overhead of the parameters passing (ν) is longer than the average method execution time (μ), e.g., $\nu>\mu$, the method calls packing factor should be decreased. In this case, the method call packing is estimated as $C_m=\alpha/\nu$.

To summarise, on pipelined applications, the μ/ν ratio is the key to choose between object and method calls packing. When $\nu<\mu$ the communication packing degree, in number of method calls per message is $C_m=\alpha/(\mu-\nu)$, and the object packing degree is decreased by the C_m factor, e.g., the number of //obj on each grain is $C_p=\gamma(\alpha+C_m\nu)/(\mu C_m)$. When $\nu>\mu$, the communication packing degree is $C_m=\alpha/\nu$ and the same C_p expression can be used to compute the number of //obj per grain.

3 SCOOPP Evaluation with the Eratosthenes Sieve

The Eratosthenes sieve is an algorithm to compute all prime numbers up to a given maximum. The original algorithm is well known; although several faster algorithms have been proposed [14][15][16], the original one is still the most adequate to illustrate relevant features in parallel algorithms, since a parallel version is intuitively obtained from the original sequential algorithm.

One simple parallel implementation is a pipelined algorithm containing all computed prime numbers, where each element filters its multiples. Numbers are sent to the pipeline on an increasing order. Each number that gets to the end of the pipeline is a prime number and is appended as a new filter. Fig.1 presents the sieve processing flow for the numbers 3, 4 and 5.

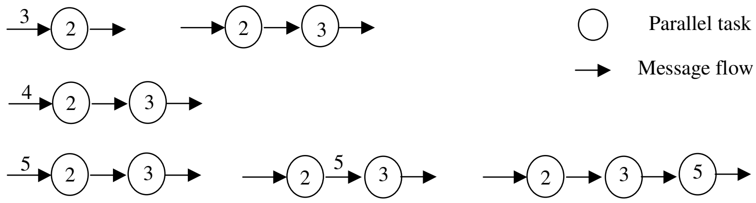

Fig. 1. Parallel sieve of Eratosthenes processing numbers 3, 4 and 5

The Eratosthenes sieve has been chosen to show the relevant features of the SCOOPP since it has a totally predictable behaviour, making it adequate to evaluate the separate impact on the execution time of each parameter and packing approach. Furthermore, it is scalable to large environments, if a large number is selected.

The Eratosthenes sieve has a large parallelism potential since each element of the pipeline (e.g., sieve filter) can be a parallel task (e.g., a parallel object), which originates a large number of fine-grained parallel tasks. It dynamically creates parallel tasks and their number is dependent of the problem size. Table 1 presents the parallelism degree of the sieve of Eratosthenes for several problem sizes.

Table 1. Parallelism degree of the Eratosthenes sieve several problem sizes

Problem size	Number of parallel tasks	Number of messages
100	24	290
1.000	167	14292
10.000	1228	762862
100.000	9591	46224072

On a naive implementation of the sieve, each parallel task has a computation to communication ratio of one integer division operation per message received, which is a too low ratio for the generality of distributed memory machines. A slightly optimised sieve was developed to increase this ratio and decrease the sieve sequential workload, which sends blocks of 10 values between sieve filters on a single method call. Each sieve filter marks the numbers that it filters and a block is merged with another block when it has more 5 values marked. This optimisation decreases the number of messages by a factor close to 10 and increases the computation to communication ratio to a value close to 10 integer divisions per message received.

The next subsection discusses how a programmer based static grain-size adaptation can increase this ratio. A second subsection shows performance results measured using the SCOOPP dynamically grain-size adaptation. Both subsections present performance results for an optimised sieve on a problem size of 100 000 values.

3.1 Programmer Based Grain-size Adaptation

This section shows how a programmer can adapt the grain-size of the sieve to improve performance on several platforms. It presents the impact of the grain-size choices on the number of //tasks and inter-//task messages. Finally, it presents the execution times of the sieve for a number of grain-size choices and analyses the impact of grain-size choices on the tested three platforms.

To adapt the grain-size in the sieve algorithm a parallel programmer may merge sieve filters into a single parallel object and/or pack several parallel objects into a single grain (e.g., a parallel task). Merging filters into a //obj requires some code rewrite, while packing //obj into a grain is less demanding: minor code modifications, mainly to adapt the load distribution policy to perform a block distribution. Merging filters into a //obj removes overheads of intra-grain object creation and method calls, leading to lower execution times (e.g., sequential workload). However, it requires complex code to support dynamic grain-size modifications.

Both approaches adapt the computational grain-size, increasing the average number of operations per received value on each //task (e.g., //task computation to communication ratio) and reducing the overall number of //tasks. On the sieve, this number of operations is directly proportional to the number of filters on each //task and is hereafter referred to as the //task computation granularity, in number of filters per parallel task. However, this increase may not lead to an acceptable performance, namely there may be not enough //tasks and the sieve may generate an excessive number of messages. Packing several method calls into a single message reduces the messages traffic, decreasing the communication overhead. On the optimised sieve under study, the number of values per message is tenfold the number of method calls per message, since each method call sends a block of 10 values, and is hereafter referred to as the inter-//task communication granularity.

Table 2 presents the number of parallel tasks and inter-tasks messages required to compute the prime numbers up to 100 000, for several //task computation and communication granularities. The grain-sizes values were selected to show representatives values of the sieve execution times.

Table 2. Sieve parallelism degree for several computation and communication granularities

Task computation granularity (filters per //task)	// tasks	Inter-tasks messages: Inter-tasks communication granularity (values per message) 10	50	100	500	1000
1	9591	4 894 536	1 005 717	518 063	118 860	67 100
6	1599	845 518	174 272	89 192	19 569	10 856
25	384	236 692	45 144	24 013	6 364	3 354
100	96	72 750	16 175	8 318	1 873	889
400	24	21 406	4 678	2 395	526	282
1600	6	8 572	1 802	915	194	102
6400	2	5 480	1 110	558	115	59
9591	1	0	0	0	0	0

Fig.2 presents the sieve execution times as a function of both the computation and communication granularities. These figures present the execution times on 4 and 7 cluster nodes, on 4 and 16 PowerXplorer nodes and on 14 and 56 MultiCluster nodes. On these experiments the measured values were obtained by using one sieve filter per //obj and grain packing was performed by packing several //obj into a single //task. The MultiCluster can not run sieves with grains smaller than 3 sieve filters, due to memory space limitations. All graphs are scaled to the sieve execution time on a single node.

On all these targets platforms the computation granularity has a strong impact on the sieve performance: when the computation grain-size is too fine or too large the performance penalties are considerably heavy. Too fine grains can lead to a large number of //tasks and the associated overhead costs; too large grains may not use all the available processing nodes.

Communication grains also have an impact on the overall performance: on smaller systems, fine grains (short messages) introduce a penalty, since they generate an excessive number of messages between pairs of nodes; on large systems, shorter and more frequent messages favour load balancing and reduce start-up times.

These results show how relevant is the right choice for both the computation and communication grain-size. However, they also show how time consuming a programmer based approach can be due to the dynamic nature of the parallel tasks of the sieve; it requires long experimental work (to test a wide range of computation and communication grain-sizes) and/or a deep analysis of both the algorithm and target platform features.

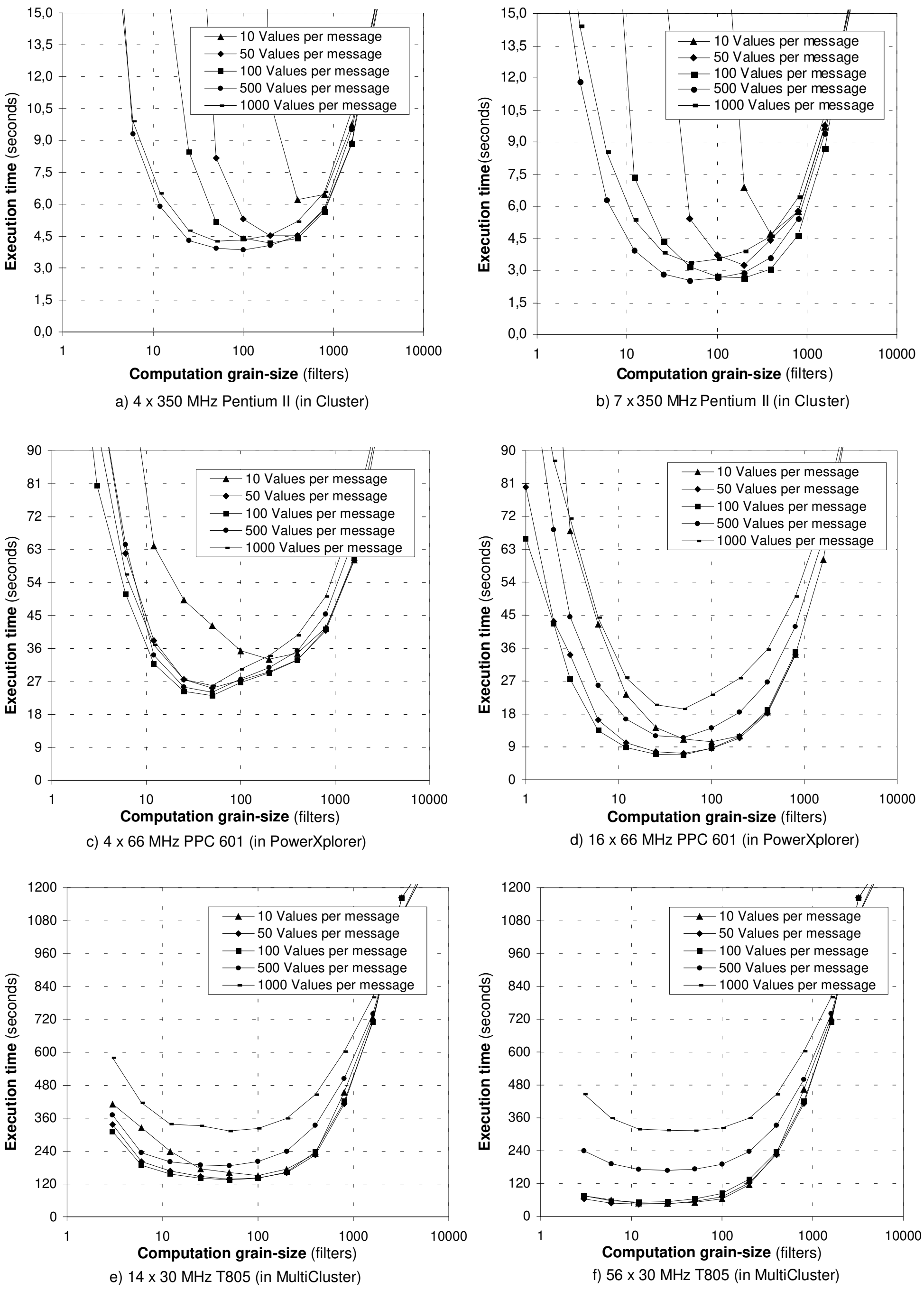

Fig. 2. Sieve execution times for a programmer based grain-size adaptation

3.2 Dynamic Grain-size Adaptation

One of the main goals of SCOOPP is dynamic scalability through grain packing. To tests de effectiveness of the SCOOPP granularity control, an evaluation of the ability to dynamically pack sieves filters was performed. This evaluation compares the lowest execution times – experimentally measured in the previous section - with the execution times obtained by running the sieve on several target platforms, without any code change and relying on the granularity control mechanisms in the SCOOPP run-time system.

Table 3 summarises the measures above mentioned. On the SCOOPP strategy, the computation and communication grain-sizes were obtained by computing the number of //obj on each //tasks and the number of method calls on each message, according the expressions on section 2.3. A circular load balancing strategy spreads grains through the nodes and can place several grains per node.

The P column shows the number of processors used for each test. For both the programmer based and SCOOPP grain-size adaptation several parameters are given: T is the execution time in seconds; Sp is the speedup obtained comparing with the same sieve on a single node; γ is the number of grains placed on each node; C_p is the degree of the computation packing, in number of //obj (filters) per //task (e.g., the computation grain-size) and C_m is the degree of the communication packing, in values per message (e.g., the communication grain-size). C_m is tenfold the number of method calls per message, since each method call sends a block of 10 values. On the SCOOPP methodology C_p and C_m are mean values, since they are computed dynamically and change during run-time.

The SCOOPP methodology results also include 3 columns with the estimated parameters, in microseconds: the remote method call latency (α), the overhead of the method parameters passing (ν) and the average method execution time (μ). The latter two parameters are also mean values.

Table 3. Comparing sieve execution times: programmer based and SCOOPP

		Programmer based					SCOOPP							
	P	T	Sp	γ	C_p	C_m	T	Sp	γ	C_p	C_m	α	ν	μ
Cluster	4	3.86	3.8	24	100	500	3.96	3.7	28	86	560	500	10	5
	7	2.52	5.9	27	50	500	2.66	5.6	21	65	560			
PowerXplorer	4	23.2	3.9	48	50	100	28.0	3.2	19	126	50	300	72	18
	16	6.9	13.0	12	50	100	8.0	11.3	12	50	50			
MultiCluster	14	135.6	9.6	14	50	100	162.3	8.0	21	34	20	530	82	440
	56	44.5	29.2	14	12	50	44.8	29.0	11	16	20			

These results show the effectiveness of the SCOOPP methodology to scale the sieve application on several target platforms. The methodology was able to dynamically increase grain-sizes to obtain speedups of the same order of magnitude as a programmer-based approach. Moreover, execution times obtained through the SCOOPP methodology are often in a 20% range of the optimal values, showing that this methodology successively removes most of the parallelism overheads. The remaining overhead is usually due to a choice of a too large or too small number of grains. However, removing this overhead requires the knowledge of the full number of //task or some guessing through experiments, as the ones performed on the previous section. These alternatives increase development costs and are not feasible on applications where the number of //tasks is strongly dependent on input data.

When computation and communication grain-sizes are controlled through packing (both object and method calls packing) the total number of created objects and method calls remains the same. To reduce this sequential workload - due to the object oriented paradigm – the programmer can "pack" by merging several //obj into a single //obj (e.g., pack several filters into a single //obj) and by grouping blocks of values on a single method call. Fig.3a and 3b show the impact of merging several filters into a single //obj and increasing the block size on method calls. The graphs show execution times on a single cluster node and the ideal execution time on 4 cluster nodes, for several computation and communication grain-sizes.

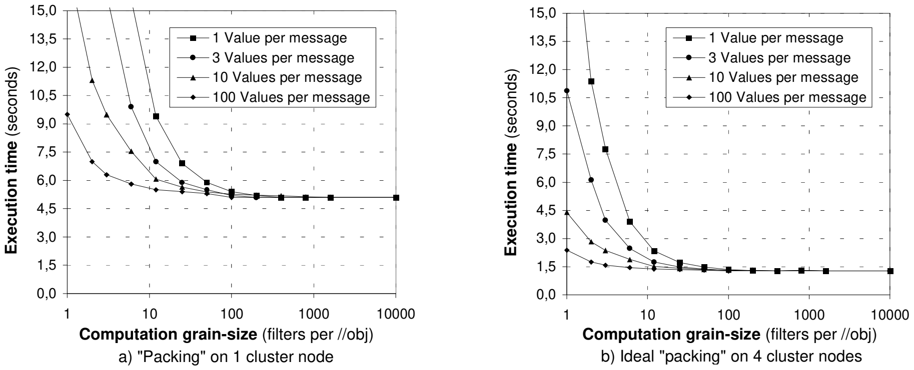

Fig. 3. Execution times for partially optimised sieves through method call and object merging

When these optimisation approaches are followed to supply SCOOPP with pre-optimised parallel versions, SCOOPP is also able to improve the overall performance. Fig.4a presents the times obtained on programmer partially optimised sieves, with communication grain-sizes of 10 and 100 values per message; Fig.4b shows their behaviour on the SCOOPP system.

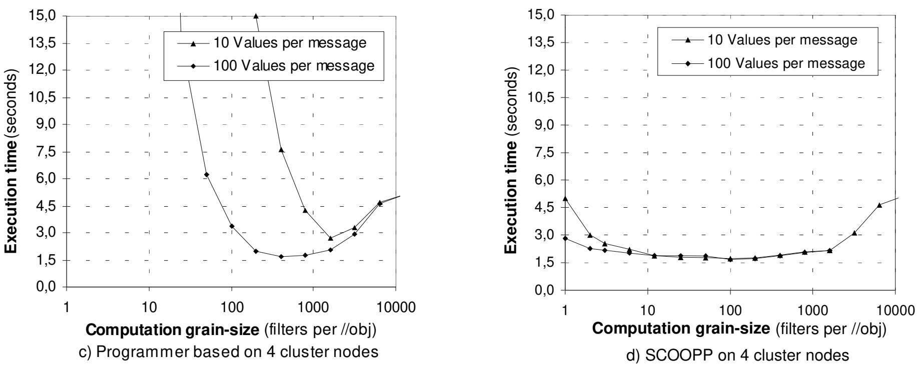

Fig. 4. Programmer based and SCOOPP implementation on partially optimised sieves

These execution times show that the SCOOPP values are very close to the “ideal” ones in Fig.3b. These results reinforce the effectiveness of the SCOOPP methodology, showing that SCOOPP can efficiently scale pipelined applications, even when these are previous and partially optimised by the programmer.

Conclusion

The commercial success of massively parallel systems was slowed down mainly due to the lack of adequate tools to support automatic mapping of the applications into distinct target platforms, without significative loss of efficiency. The overhead on programmers was too high and the available tools were inefficient. SCOOPP attempts to overcome these limitations: it provides dynamic and efficient scalability of object oriented parallel applications across several target platforms, packing grains and messages, without any code modification.

The presented results show the effectiveness of the SCOOPP methodology when applied to pipelined applications on several target platforms. The methodology is able to dynamically increase grain-sizes and to obtain speedups of the same order of magnitude as a programmer-based approach. Moreover, execution times obtained through the SCOOPP methodology are often in a 20% range of the optimal values, showing that this methodology successively removes most of the parallelism overheads.

Programmer based grain-size adaptation is not a competitive alternative to SCOOPP, it requires a wide range of tests on each target platform and each test is highly time consuming (as presented in Fig.2).

The performance penalties imposed by SCOOPP have a low impact on application execution time, and they are mainly due to the run-time requirements to estimate the application dependent parameters to adapt the computation and communication grain-sizes. A static adaptation can provide the correct grain-size at the beginning of the running, but a dynamic strategy requires some time to evaluate the application features and to react accordingly.

Dynamic scalability of the parallel code version largely overcomes this small performance cost. It is the most promising approach to scale applications where task granularity is strongly dependent on input data. When compile time estimates of task granularity are not accurate, it may decrease the cost of the parallel code development and improve the code reutilization on multiple target platforms.

Current work includes development of packing policies for static and dynamic object trees, and applied to less controlled application environments (such as computer vision applications).

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] High Performance Fortran Forum: HPF language specification, Technical Report CPRPC-TR92225, Center for Research on Parallel Computation, Rice University, Tex., (1993)

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

[5] Andersen, A.: A General, Fine-Grained, Machine Independent, Object-Oriented Language, ACM SIGPLAN Notices, Vol. 29(5), May (1994)

[6] Sobral, J., Proença, A.: Dynamic Grain-Size Adaptation on Object-Oriented Parallel Programming - The SCOOPP Approach, Proceedings of the 2nd Merged IPPS/SPDP 1999, Puerto Rico, April (1999)

[7] 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)

[8] Sobral, J., Proença, A.: A Run-time System for Dynamic Grain Packing, Proceedings of the 5th International EuroPar Conference (Euro-Par'99), Toulouse, France, September (1999)

[9] 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, Vol. 2(3), July (1991)

[10] Goldstien, S., Schauser, K., Culler, D: Lazy Threads: Implementing a Fast Parallel Call, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)

[11] Karamcheti, V., Plevyak, J., Chien, A.: Runtime Mechanisms for Efficient Dynamic Multithreading, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)

[12] Taura, K., Yonezawa, A.: Fine-Grained Multithreading with Minimal Compiler Support – A Cost Effective Approach to Implementing Efficient Multithreading Languages, Proceedings of the ACM SIGPLAN Conference on Programming Language Design & Implementation (CPLDI'97), Las Vegas, July (1997)

[13] Lopez, P., Hermenegildo, M., Debray, S.: A Methodology for Granularity Based Control of Parallelism in Logic Programs, Journal of Symbolic Computation, Vol. 22, (1998)

[14] Pritchard, P.: Linear Prime-Number Sieves: A Family Tree, Science of Computer Programming, Vol. 9, (1987)

[15] Xuedong, L.: A Practical Sieve Algorithm Finding Prime Numbers, Communications of the ACM, Vol. 32(3), (1989)

[16] Dunten, B., Jones, J., Sorenson, J.: A Space-Efficient Fast Prime Number Sieve, Information Processing Letters, Vol. 59, (1996)