Medium Grained Execution in Concurrent Object Oriented Systems*

Laxmikant V. Kale   Sanjeev Krishnan
Department of Computer Science,
University of Illinois, Urbana-Champaign.
email : {kale,sanjeev}@cs.uiuc.edu

Abstract

We present arguments for the adequacy of medium grained execution for concurrent object oriented systems such as Charm++ on current stock multicomputers. We briefly describe Charm++ and present performance data for two of its basic primitives. We argue that although grainsize decisions must be made by the programmer, they remain portable across different machines and with different numbers of processors.

1 Introduction

Concurrent object oriented systems can be broadly classified into two categories. The first category comprises systems in which all objects can execute in parallel on remote processors. Such systems are usually fine grained because every object can be potentially fine grained, hence message passing and object creation overheads must be low, in order to balance computation and communication.

The second category comprises systems which distinguish between sequential and parallel objects. Explicit grainsize control is usually needed to create sequential computations of sufficient size such that message passing overheads are amortized. These sequential computations are coordinated by parallel objects. Charm++ [KK93] is such a medium grained concurrent object oriented system. In the next few sections we present arguments for the advantages of medium grained systems such as Charm++ and compare it to the first category.

2 A brief description of CHARM++

Charm++ is an explicitly parallel language consisting of C++ with a few extensions for specification of parallel objects and communication between parallel objects. The execution model of Charm++ is message driven, thus helping one write programs that are latency-tolerant.

Charm++ provides a clear separation between sequential and parallel objects. This is desirable in order to make the cost of an operation (expensive remote calls as opposed to simple local calls) clear to the programmer, and also to emphasize the split-phase nature of Charm++ calls. Moreover, separating the parallel part of the algorithm (which coordinates the sequential computations) is helpful for algorithm design.

*This research was supported in part by the National Science Foundation grants CCR-91-06608 and CCR-90-07195.

The language supports multiple inheritance, dynamic binding, overloading, strong typing, and reuse for parallel objects. Charm++ provides specific modes for sharing information between parallel objects. Extensive dynamic load balancing strategies are provided. It is based on the Charm parallel programming system [Kal90, SK90], and its runtime system implementation reuses most of the runtime system for Charm.

3 Timings for basic primitives in CHARM++

We measured the times for two basic operations in Charm++, viz. intra node object creation and round trip messaging. The times for the CM5 are in Table 1.

Primitive	Intra Node Object Creation	Round Trip Messaging
Times (μs) on the CM5	150	645

Table 1: **Times (in microseconds) for two Charm++ primitives on the CM/5.**

Round trip times were measured for a user program message size of 4 bytes. The messages were sent from user program to user program and back, hence the times represent latencies as seen by the application program. It was observed that the Charm++ overhead itself was about 200 μs, the rest being overhead due to the asynchronous messaging software and network latencies on the CM5. The object creation time includes the time to allocate and fill in a message for creation of an object, time spent by the runtime system in scheduling the message (i.e. enqueueing and dequeueing it in the scheduler's queue, etc), creation of the actual object, and execution of the code for initializing it, in addition to the routine overheads of checking for the presence of remote messages, etc.

We observe from the table that the times are much smaller compared to Unix (or operating system level) processes, which are typically of the order of a few tens to hundreds of milliseconds. Thus one can create more than 6000 chares per second on each processor. The Charm runtime system is being optimized and one can expect a further reduction in these times. Despite this, however, the times in Charm are substantially higher than the results reported for fine grained systems such as ABCL [YMY92] and the J-Machine [D$^+$89], which are usually of the order of a few tens of microseconds. Thus Charm++ may be said to be a medium grained system.

4 Why medium grained systems ?

Medium grained systems are currently a good choice for a programming system because :

Implementation : Medium grained systems can be easily implemented in a portable manner on stock commercial multicomputers. This is because they can take advantage of existing message passing software (such as CMMD on the CM5) and compilers (such as a C++ compiler) on commercial multicomputers. Typically they consist of a translator integrated with a runtime system. Because of easy portability, such systems can be quickly ported to new machines.

Grainsize control : The execution time of a parallel program as a function of grainsize can be approximated by the curve in figure 1.

The communication overhead of a system is proportional to $\frac{1}{g}$ where g is grainsize. For small

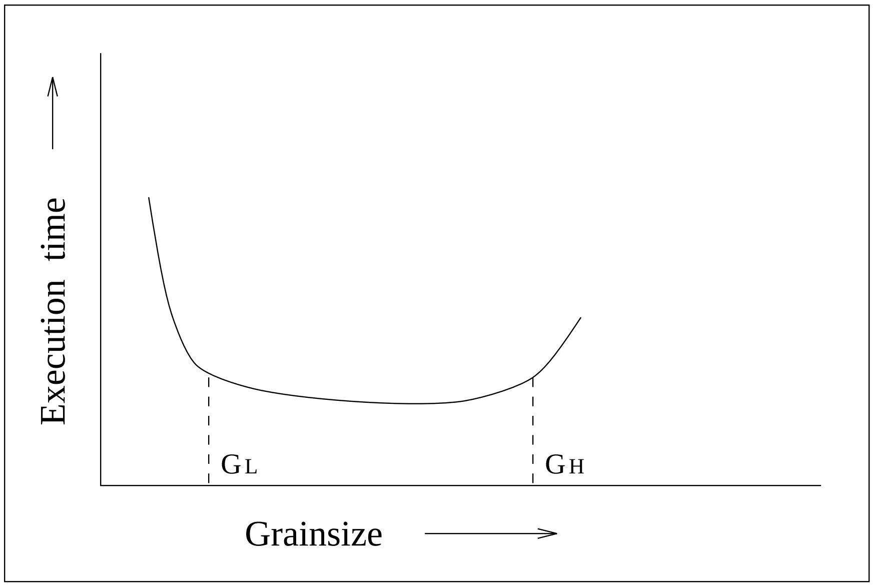

Figure 1: Parallel overhead of a system as a function of grainsize.

grainsizes communication overhead and hence execution time decreases rapidly with an increase in grainsize. The large, almost flat region at the bottom of the curve can be expected because the communication overhead becomes a small fraction of the overall execution at medium to large grainsizes. Thus although the overhead continues to decrease with increasing grain size, its impact on execution time is insignificant. When grainsize increases beyond point G_H on the curve, loss of parallelism causes processor cycles to be wasted and hence there is an increase in execution time.

When the number of processors increases, G_H moves to the left because more processors remain idle for a particular grainsize. However, G_L can be expected to remain approximately at the same point, because it depends on the value of the grain size in relation to message passing overhead.

With the above in mind, the programmer can set a fixed grainsize without affecting performance portability for varying numbers of processors and different machines, because :

- by using the smallest grainsize (corresponding to point G_L on the curve) which sufficiently amortizes overheads, we are assured of good performance with a small as well as a large number of processors. With a large number of processors this is appropriate because there is sufficient parallelism. For a small number of processors we will get good performance because the overheads are small compared to the actual computations (in fact, a larger number of small grained computations can be better load balanced).

- techniques such as message driven execution as used in Charm++ can mitigate variability of communication delays on different machines. With message driven execution a processor can adaptively execute useful computations while waiting for arrival of a message, thus effectively overlapping communication with computation. For most computations this will ensure that the communication latency (the time for data to physically move across the network) is effectively hidden and does not impact execution times. Thus the communication overhead that determines G_L is mainly the software overhead. Even though the processor speeds are different, the number of instructions contributing to the software overhead is the same, hence the ratio of software overhead to grainsize is approximately a constant across machines. Hence the performance of the program is portable across a wide range of machines without change in grainsize.

Medium grained systems are sufficient for a large class of applications. By involving the programmer directly, medium grained systems can yield better performance, since the programmer can tune the grain size to optimize parallelism and reduce communication. In contrast, automatic grainsize control techniques are not sufficient to yield optimal grainsize decisions in all cases. Moreover, compiler generated transformations may affect performance in unpredictable ways. Finally, a system such as Charm++ can be used as a portable back end even when automatic grainsize control techniques are deemed sufficient.

The disadvantages of medium grained execution are increased programmer involvement, and its unsuitability for inherently fine-grained applications. Research in fine grained systems and automatic grainsize control may remove these disadvantages in the future.

References

[D$^+$89] W. Dally et al. The J-Machine : A fine grained concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.

[Kal90] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[KK93] L. V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993. To Appear.

[SK90] W. W. Shu and L. V. Kale. Chare Kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1990.

[YMY92] M. Yasugi, S. Matsouka, and A. Yonezawa. ABCL/onEM4 : A new software / hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *6th ACM International Conference on Supercomputing*, July 1992.