2.16 Parallel Implementation

The solution of many problems in computer algebra requires a large amount of computation time and memory space. In order to overcome this difficulty, various researchers have pursued the development of highly parallel algorithms [42, 24, 3] and their efficient implementation. For a general overview of parallel computer algebra, see for example [11, 44, 23, 20, 37].

This section gives an overview on the fundamental components required for the design and implementation of parallel *irregular applications* [16] which dynamic behavior is difficult to predict. The major problem encountered in the parallelization of many computer algebra algorithms is related to this irregularity: intermediate expressions swell [10] may grow and shrink in a seemingly random manner.

Implemented on top of an *operating system* that provides software tools to exploit parallel architectures, a parallel computer algebra system has two tackle three challenges that are related to the parallelization of any general irregular application. First, many objects are represented by pointer-linked data structures that are dynamically allocated from a heap and that are automatically reclaimed by a garbage collector; thus *data management* is challenging. Second, it is hard to determine in advance the number of tasks and their grain sizes; thus *task scheduling* is a critical point to reach efficient executions. Furthermore, algorithmic structures are often so complicated that existing parallel language frameworks are not suitable; thus *language design* is also challenging.

Parallel architectures and operating systems supports. The main challenge of parallel computing is to efficiently exploit independent hardware units (loosely coupled processors) with limited resources (memory, network) that are hierarchaly organized. A general parallel architecture is composed of SMP[1] machines (each of them has several processors with cache memory and a high speed bus to access memory banks) connected by a network that permits access to remote memory. Due to the hierarchy, the costs of memory access are not uniform and usually define the *grain of the architecture*. This is a key-point relevant to the difference between sequential and parallel computers: in the latter there are potential large overheads in memory accesses[2].

In order to exploit these hardware resources, the operating system (Unix) provides interfaces for three main features. First, an access to the network using point-to-point communication (*e.g.* BSD socket, PVM, MPI) or collective communication (*e.g.* PVM, MPI, BSP). Second, it permits to exploit the processors inside a SMP machine using *lightweight processes* or *threads* (*e.g.* Posix Thread, OpenMP) with operations for the synchronization (*e.g.* semaphore, mutex, monitor). Third, a shared memory (an address space inside a Unix process shared by threads or through a distributed shared memory library) or a remote memory

[1] Symmetric multi-processors.

[2] The performance of the network ranges from Ethernet link at 10Mbit/s to high-performance dedicated network as found in SGI Origin 2000 super-computer (about $1Gbit/s$).

access library (Cray SHMEM, Scalable Coherent Interface, Memory Channel interface).

Data management. Most computer algebra programs rely on data structures that are dynamically allocated on the heap, that are referenced via pointers, and that are automatically reclaimed by a garbage collection mechanism when they are not referenced any more.

On systems with efficient shared memory support (such as most small to medium-scale SMP machine), pointers can be freely exchanged between tasks and access to the referenced data structures is possible at almost no additional cost[3]. In order to allow multiple tasks to allocate heap memory simultaneously and efficiently, methods are used that do not enforce serialization by a central lock [27]. Since memory is allocated by concurrent tasks at a higher total rate, also the garbage collector is parallelized in order to reclaim memory at a corresponding speed [28, 21].

On most large-scale multiprocessors and computer networks, several distributed garbage collection techniques have been studied [33]. The efficient use of global references relies on the ability to ensure locality of data access. This introduces an overhead due to the serialization (marshaling, unmarshaling) during communication, and to the distributed protocol to maintain references onto data. Thus, efficiency of execution relies on both the scheduling and the user program which should be designed carefully [4]. Moreover, due to copies of data to increase speed of memory access, scheduling algorithms should be designed to control the potential memory exhaustion.

Task management: mapping and scheduling. The operating system schedules multiple threads of computation in order to ensure a fair sharing of resources (CPUs) among several applications (or users). Mapping the tasks dynamically generated by an application onto threads of an operating system is not reliable: the cost of creation, the memory resources needed by a thread may cause considerable execution overhead. Therefore, a scheduling algorithm maps these tasks onto a limited set of threads. The goal is to minimize the completion time of an execution while preserving the used memory space. Such a task is a small descriptor of a call to a function (also called virtual threads in other works [30]). Due to the dynamic evolvement of the load on architectures that may be shared by several users, and also irregularity of most computer algebra applications, this schedule of tasks is computed on-line.

On-line scheduling algorithms are mainly based on the management of a task queue: when a processor becomes idle, it takes a ready task if any in the queue. When overheads related to the computation of the schedule and communications are neglected, such a popular strategy ensures provable performances on architectures with identical processors: asymptotically close to the optimal for

[3] At most at the cost of cache miss or false sharing due to a bad organization of data structure.

[4] Note that, at a finer grain, this should be consider for programs on SMP architecture in order to efficiently re-use data in cache memory.

fine-grain computations [5] and at most at a factor two from the optimal in the general case [18]. To decrease concurrency on the queue of tasks, it is often implemented in a distributed way following a *work-stealing* scheme. Each processor manages its own queue and attempts to steal work from other processors only when it becomes under-loaded [34]. The processor to steal can be chosen according to cpu and memory load informations of the machine. For instance, in DSC this feature appeared useful to efficiently solve sparse linear systems where the matrix requires a huge amount of memory [12].

The main problem encountered in the use of such a scheduling is related to memory space exhaustion [1, 4]. This exhaustion cannot be avoided for a general multithreading model [5] where synchronization operators, such as semaphores, are unpredictable and forbid a serial execution. However, if a correct serial execution order is a-priori known, managing a priority queue according to this order enables to bound the memory space with respect to the one required by the serial execution. Such a serialization restricts the programming model: fully-strict computations in Cilk [5], nested computations in Nesl [31], planar graphs [4] and non-blocking tasks in Athapascan [15].

Serialization versus parallelization is also related to time efficiency concerning the tuning of the granularity: how to decide if a computation should be splitted into subsequent concurrent tasks or not ? Assuming the correctness of a depth-first serial execution, Cilk implementation answers this question by compiling task creation into a local function call with a very small overhead; effective creation of a closure occurs only when a processor receives a work request from another one. Since idle time is only related to the parallel time on an unbounded number of processors, only few steal operations are performed for program having a small critical path. Cilk implementation has proven the practical performances of such a strategy for fine grain fully-strict programs [25] that encompasses the fork-join programming model.

Beyond granularity, reducing synchronization between processors and remote data access is also crucial for time efficiency.

Furthermore, due to the magnitude of the ratio between local and remote memory access costs on a distributed architecture, additional informations about relationships between data access and threads can also be used. In parallel numerical computation, scheduling threads according to this information has been successfully used in linear algebra. PYRROS [17] performs a clustering of tasks and has been used for matrix factorization; METIS [26], based on data dependency graph partitioning, is widely used for sparse matrix-vector product. Those informations can be synthesized at runtime from a macro-data-flow analysis [14, 35, 15].

The computation of fine schedules introduces large overheads which may appear suited to only a specific class of programs. Then, various approaches are distinguished. Cilk argues for a restricted model of parallelism that enables a low-overhead runtime schedule with provable performances. Most languages, like Java, provide only local scheduling; global scheduling has to be implemented by the user. Between both, some languages like OPEN MP provide code annotations to tune the schedule.

Parallelism expression and languages. In the early days, parallel programming paradigms where based on explicit synchronization: message passing, fork/join. Current trends in parallel programming are to abstract these low level details by relying on a high level language where parallelism is explicit but independent from the target architecture; synchronizations become implicit using compilation techniques (data flow analysis,...). The underlying runtime support is based on operating systems tools and includes global scheduling algorithms of tasks. For most of those high-level languages, dependencies between tasks (*e.g.* synchronizations) are synthesized at runtime from a macro-data flow analysis [14, 35, 15].

Nevertheless, experiments have shown that it is difficult to have good efficiency using automatic parallelization. Therefore, code annotations have been defined in order to specify a good *grain of parallelism* for both data and computations. Likewise the functional idea of algorithmic "skeletons" to encapsulate typical parallelization patterns (by higher order functions or by program templates) is well suited for computer algebra where many parallel algorithms are based on related mathematical concepts (e.g., the wide-spread use of modular methods for parallelization).

The functional/dataflow programming model has had considerable influence on the development of various systems for computer algebra. Functions that are spawned as separate tasks (whose handle can be used to wait for the termination of the task and get its result) have their origin in the "future" construct of MultiLisp [19]. Various parallel computer algebra methods have also successfully applied the functional idea of using non-strict data structures [2] for building "streams" of messages by which tasks may continuously communicate with each other.

Software Systems. The two main problems from building a parallel computer algebra system are to include parallel features (externalization of data representation, thread safety of libraries for multi-threaded environment) and to define a parallel programming.

To bypass this problems first parallel computer algebra systems have been designed on the integration of a parallel runtime support into a modified computer algebra library (*e.g.* PARSAC [27] and PACLIB [21], respectively pD [38] for shared memory machines and SAC-2 [40], DSC [13, 7], PAC/Givaro [36, 23], Sturm [22], REDUCE [32], DTS [6], CABAL [20], and CALYPSO [20] for distributed systems).

The integration of complete computer algebra systems such as Mathematica or Maple faces the practical problem that (due to the commercial nature of most systems) access to the internal structure of the kernel is prohibited. The exchange of data structures between different kernels is bound to a vendor-supplied communication protocol or even has to pass the standard input/output interface. Only MuPad [29], an experimental version of Maple [20] based on a modified kernel, and a share package for GAP integrate into the kernel a parallel programming support based on the message passing model. Axiom and its compiler Axiom-XL [43] permits linkage with C functions, and gives a partial access to internal structure; this is used at ETHZ–Zürich for integrating parallelism into Axiom.

More frequently, computer algebra kernels are embedded as "co-processors" into the framework of a parallel language or system: the parallel algorithm is expressed in the terminology of this coordination layer that distributes the actual computations among the kernels (||MAPLE|| [41], Linda-Maple [9]). In other approaches, the computer algebra system itself serves as the parallel programming environment; via an external interface it communicates with the underlying parallel runtime system for the distribution of tasks and data (Maple/Linda–Sugarbush [8], Distributed Maple [39]).

	DSC	PARSACS-2	SBSH	PCLBSTM	PCGVR	||MAPLE||	μP
ARCHITECTURE							
- dist. memory	•	•	•	•	•	•	(•)
- shared. memory		•	•	•	•	•	•
- heterogeneous	•		•		•		
DATA MODEL							
- message passing		•		•			•
- virtually shared			•		•		(•)
TASK MODEL							
- heavy processes	•						
- multi-threading		•		•	•		•
-SYNCHRONIZATION							
explicit	•	•	•	•			
implicit			•		•	•	•
LOAD-BALANCING							
- distributed		•			•		
- load indicator	•	•	•	•	•		
- user info.	•	•	(•)		•		

Table 1. Abstract of some characteristics of softwares for parallel computer algebra. SBSH means Sugarbush. PCLBSTM means PACLIB/STURM and PCGVR PAC/GIVARO.

Conclusions. Major problems that appears in the design and implementation of parallel computer algebra systems (on-line scheduling of tasks, distributed garbage collection) are studied in the more general context of modern parallel programming languages.

Also, parallel computer algebra focus nowdays on two directions. First, a development effort has to be done to integrate a modern parallel language in a general purpose computer algebra system with its huge libraries of complex algorithms. Second, active research should be done in the design of efficient and portable parallel algorithms for more complex problems.

Thierry Gautier (INRIA, LMC-IMAG), Hoon Hong (NCSU), Jean-Louis Roch (LMC-IMAG), Gilles Villard (CNRS, LMC-IMAG), Wolfgang Schreiner (RISC-Linz)

References

1. Arvind and David E. Culler. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architectures*, pages 141–150, Honolulu, Hawai, May 1988.

2. Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel programming. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.
3. D. Bini and V. Pan. *Polynomial and matrix computations.* Birkhäuser, 1994.
4. Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space efficient scheduling of Parallelism with Synchronization Variables. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures.* ACM Press, June 1997.
5. Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.
6. T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel. Distributed symbolic computation with dts. In *Proc. of IRREGULAR'95, Lyon, France*, volume 980 of *LNCS*. Springer-Verlag, September 1995.
7. K.C. Chan, A. Diaz, and E. Kaltofen. A Distributed Approach to Problem Solving in Maple. In R.J. Lopez, editor, *Maple V: Mathematics and its Application, Proceedings of the Maple Summer Workshop and Symposium (MSWS'94)*, pages 13–21, Boston, 1994. Birkhäuser.
8. B. Char, J. Johnson, D. Saunders, and A.P. Wack. Some experiments with parallel Bignum arithmetic. In *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, Lecture Notes Series in Computing – Vol. 5, pages 94–103. World Scientific Publishing, 1994.
9. B.W. Char. A users's guide to sugarbush – parallel Maple through Linda. Technical Report MCS-94-01, Drexel University, Department of Mathematics and Computer Science, 1994.
10. R. Corless, D. Jeffrey, M. Monagan, and Pratibha. Parallel symbolic computing. *IEEE Computer*, 23(4):427–443, April 1997.
11. J. Della Dora and J. Fitch, editors. *Computer Algebra and Parallism.* Academic Press, 1989.
12. A. Diaz, M. Hitz, E. Kaltofen, A. Lobo, and T. Valente. Process scheduling in DSC and the large sparse linear systems challenge. *Journal of Symbolic Computation*, 19(1–3):269–282, 1995.
13. A. Diaz, E. Kaltofen, K. Schmitz, and T. Valente. Dsc a system for distributed computation. In M. Watt, editor, *SSAC'91*, pages 324–333. ACM Press, 1991.
14. J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.
15. François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mahtias Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *IEEE Annual Conference on Parallel Architectures and Compilation Techniques, PACT'98*, Paris, France., October 1998.
16. T. Gautier, J.L. Roch, and G. Villard. Regular versus irregular problems and algorit hms. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structure d Problems*, volume 980 of *LNCS*, pages 1–25, 1995.
17. A. Gerasoulis and T. Yang. PYRROS : Static Task Scheduling and Code Generation for Message-Passing Multi-Processor. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, 1992.
18. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.
19. R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 4(7):501–538, 1985.

20. M. Hitz and E. Kaltofen, editors. *Second International Symposium on Parallel Symbolic Computation (PASCO'97)*. ACM, 1997.
21. H. Hong, A. Neubacher, and W. Schreiner. The design of the SACLIB / PACLIB kernels. In A. Miola, editor, *DISCO'93, Gmunden, Austria*, LNCS 722. Springer-Verlag, 1993.
22. H. Hong, A. Neubacher, W. Schreiner, and V. Stahl. The c++ interface to the STURM multi-processor kernel. Technical Report TR 94-31, RISC Institute, Linz, Austria, 1994.
23. Hoon Hong, editor. *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, Lecture Notes Series in Computing, Linz, Austria, September 1994. World Scientific Publishing.
24. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachussets, 1992.
25. C.F. Joerg. *The Cilk system for parallel multithreaded computing*. PhD thesis, Massachussets Institute of Technology, january 1996.
26. George Karypis and Vipin Kumar. Multilevel k-way partitionning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
27. W. Küchlin. Parsac-2: a parallel sac-2 based on threads. In *Eights Int. Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Tokyo, Japan.*, LNCS 508. Springer-Verlag, 1990.
28. Wolfgang Küchlin and Nicholas Nevin. On Multi-Threaded List-Processing and Garbage Collection. In *3rd IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December, 1991. IEEE Press.
29. Metzner, Radimersky, Sorgatz, and Wehmeier. *MuPAD User's Guide to Macro Parallelism in MuPAD 1.4.1.* University of Paderborn, August 1998.
30. E. Mohr, D. A. Kranz, and Jr. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
31. Girija J. Narlikar and Guy E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 1997.
32. Winfried Neun and Herbert Melenk. *RR: Parallel Symbolic Algorithm Support for PSL based REDUCE*, 1995. Preliminary Draft.
33. David Plainfossi and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), September 1995.
34. V.J. Rayward-Smith, F. W. Burton, R.M. Fujimoto, and G.J. Janacek. Load balancing strategies for the implementation of parallel programs. Technical report, Simon Fraser University Centre for System Science, Vancouver, Canada, 1990.
35. M. Rinard. The design, implementation and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.
36. Jean-Louis Roch. An Environment for Parallel Algebraic Computation. In R. E. Zippel, editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 33–50, Ithaca, USA, May 1990. Springer, Berlin.
37. Jean-Louis Roch and Gilles Villard. Parallel Computer Algebra. Technical report, Lecture notes for a tutorial in ISSAC'97, Hawaii, July 1997.
38. Wolfgang Schreiner. A Para-Functional Programming Interface for a Parallel Computer Alge bra Package. *Journal of Symbolic Computation*, 21(4–6):593–614,

1996.
39. Wolfgang Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998.
40. Steffen Seitz. Algebraic Computation on a Local Net. In R. E. Zippel, editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 19–31, Ithaca, USA, May 1990. Springer, Berlin.
41. Kurt Siegl. ‖MAPLE‖ — A System for Parallel Symbolic Computation. In Hussein M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium (IPPS'93)*, pages 62–67, Newport Beach, California, April 14, 1993.
42. Joachim von zur Gathen. Parallel algorithms for algebraic problems. *SIAM Journal on Computing*, 13:802–824, 1984.
43. S.M. Watt et al. A First Report on the $A^\sharp$ Compiler. In *Proc. ISSAC'94*, pages 25–31, 1994.
44. R.E. Zippel, editor. *Computer Algebra and Parallism*, Lecture Notes in Computer Science, Ithaca, USA, May 1990. Springer-Verlag.