Automatic Parallelization and Scheduling of Programs on Multiprocessors using CASCH

Ishfaq Ahmad[1], Yu-Kwong Kwok[1], Min-You Wu[2] and Wei Shu[2]

[1]Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong
[2]Department of Computer Science, State University of New York at Buffalo, New York
Email: {iahmad, csricky}@cs.ust.hk, {wu, shu}@cs.buffalo.edu

Abstract†

The lack of a versatile software tool for parallel program development has been one of the major obstacles for exploiting the potential of high-performance architectures. In this paper, we describe an experimental software tool called CASCH (Computer Aided SCHeduling) for parallelizing and scheduling applications to parallel processors. CASCH transforms a sequential program to a parallel program with automatic scheduling, mapping, communication, and synchronization. The major strength of CASCH is its extensive library of scheduling and mapping algorithms representing a broad range of state-of-the-art work reported in the recent literature. These algorithms are applied for allocating a parallelized program to the processors, and thus the algorithms can be interactively analyzed, tested and compared using real data on a common platform with various performance objectives. CASCH is useful for both novice and expert programmers of parallel machines, and can serve as a teaching and learning aid for understanding scheduling and mapping algorithms.

1 Introduction

Parallel machines provide tremendous potential for high performance but their programming can be a tedious task. The software development process for parallel processing includes designing a parallel algorithm, partitioning the data and control, communication, synchronization, scheduling, mapping, and identifying and interpreting various performance measures. While an efficient implementation of some of these tasks can only be done manually, a number of tedious chores, such as scheduling, mapping, and communication can be automated.

Several research efforts have demonstrated the usefulness of program development tools for parallel processing. Essentially, these tools can be classified into two types. The first type of tools are mostly commercial tools which provide software development and debugging environments [5], [6], [10]. Some of these tools also provide performance tuning and other program development facilities [3], [11], [19]. A major drawback of some of these tools is that they are essentially simulation environments. While they can help in understanding the operation and behavior of scheduling and mapping algorithms, they are inadequate for practical purposes. The second type of tools performs some program transformation through program restructuring [7], [9], [13], [17], [22], [23], [24]. However, these tools are usually not well integrated with sophisticated scheduling algorithms.

In this paper, we describe a software tool called CASCH (Computer Aided SCHeduling) for parallel processing on distributed-memory multiprocessors. CASCH can be considered to be a super set of tools such as PAWS [19], Hypertool [23], PYRROS [24], and Parallax [17], since it includes the major functionalities of these tools at a more advanced and comprehensive level and also offers additional useful features. CASCH is aimed to be a complete parallel programming environment including parallelization, partitioning, scheduling, mapping, communication, synchronization, code generation, and performance evaluation. Parallelization is performed by a compiler that automatically converts sequential applications into parallel codes. The parallel code is optimized through proper scheduling and mapping, and is executed on a target machine. CASCH provides an extensive library of state-of-the-art scheduling algorithms from the recent literature. The library of scheduling algorithms is organized into different categories which are suitable for different architectural environments.

The scheduling and mapping algorithms are used for scheduling the task graph generated from the user program. The weights on the nodes and edges of the task graph are computed using a database that contains the timing of various computation, communication, and I/O operations for different machines. These timings are obtained through benchmarking. An attractive feature of CASCH is its easy-to-use GUI for analyzing various scheduling and mapping algorithms using task graphs generated randomly, interactively, or directly from real programs. The best schedule generated by an algorithm can be used by the code generator for generating a parallel program for a particular machine—the same process can be repeated for another machine.

The rest of this paper is organized as follows. Section 2 gives an overview of CASCH and describes it major functionalities. Section 3 includes the results of the experiments conducted on the Intel Paragon using CASCH. The last section includes a discussion of the future work and some concluding remarks.

2 Overview of CASCH

The overall organization of CASCH is shown in Figure 1. The main components of CASCH includes:

- A compiler which includes a lexer and a parser;
- A DAG (directed acyclic graph) generator;
- A weight estimator;
- A scheduling/mapping tool;
- A communication inserter;
- An interactive display unit;
- A code generator;
- A performance evaluation module.

These components are described below.

User Programs: Using the CASCH tool, the user first writes a sequential program from which a DAG is generated.

†. This research was partly supported by a grant from the Hong Kong Research Grants Council under contract number HKUST 734/96E and HKUST RI 93/94.EG06.

0190-3918/97 $10.00 © 1997 IEEE

To facilitate the automation of program development, we use a programming style in which a program is composed of a set of procedures called from the main program. A procedure is an indivisible unit of computation to be scheduled on one processor. The grain sizes of procedures are determined by the programmer, and can be modified with CASCH.

The control dependencies can be ignored, so that a procedure call can be executed whenever all input data of the procedure are available. Data dependencies are defined by the single assignment of parameters in procedure calls. Communications are invoked only at the beginning and the end of procedures. In other words, a procedure receives messages before it begins execution, and it sends messages after it has finished the computation.

Lexer and Parser: The lexer and parser analyze the data dependencies and user defined partitions. For a static program, the number of procedures are known before program execution. Such a program can be executed sequentially or in parallel. It is system independent since communication primitives are not specified in the program. Data dependencies among the procedural parameters define a macro dataflow graph.

Weight Estimator: The weights on the nodes and edges of the DAG are inserted with the help of an estimator that provides timings of various instructions as well as the cost of communication on a given machine. The estimator uses actual timings of various computation, communication, and I/O operations on various machines. These timings are obtained through benchmarking using an approach similar to [19]. Communication estimation, which is obtained experimentally, is based on the cost for each communication primitive, such as *send*, *receive*, and *broadcast*.

DAG Generation: A macro dataflow graph, which is generated directly from the main program, is a directed graph with a start and an end point. Each node corresponds to a procedure, and the node weight is represented by the procedure execution time. Each edge corresponds to a message transferred from one procedure to another procedure, and the weight of the edge is equal to the transmission time of the message. When two nodes are scheduled to a single processor, the weight of the edge connecting them becomes zero. The execution time of a node is obtained by using the estimator. The transmission time of a message is estimated by using the message start-up time, message length, and communication channel bandwidth.

Scheduling/Mapping Tool: A common approach to distributing workload to processors is to partition a problem into *P* tasks and perform a one-to-one mapping between the tasks and the processors. Partitioning can be done with the "block", "cyclic", or "block-cyclic" pattern [10]. Such partitioning schemes are suitable for problems with regular structures. Simple scheduling heuristics such as the "owner compute" rule work for certain problems but could fail for many others, especially for irregular problems, as it is difficult to balance load and minimize dependencies simultaneously. The way to solve irregular problems is to partition the problem into many tasks which are scheduled for a balanced load and minimized communication. In CASCH, a DAG generated based on this partitioning is scheduled using a scheduling algorithm. However, one scheduling algorithm may not be suitable for a certain problem on a given architecture.

CASCH includes various algorithms (see Figure 1) which are suitable to various environments. Currently, CASCH includes three classes of algorithms [2]: the *UNC (unbounded number of clusters)*, the *BNP (bounded number of processors)*, and the *APN (arbitrary processor network)* scheduling algorithms. The UNC scheduling algorithms, which are mostly based on clustering techniques, are designed for scheduling with unlimited number of processors. The BNP scheduling algorithms, which are based on the *list*

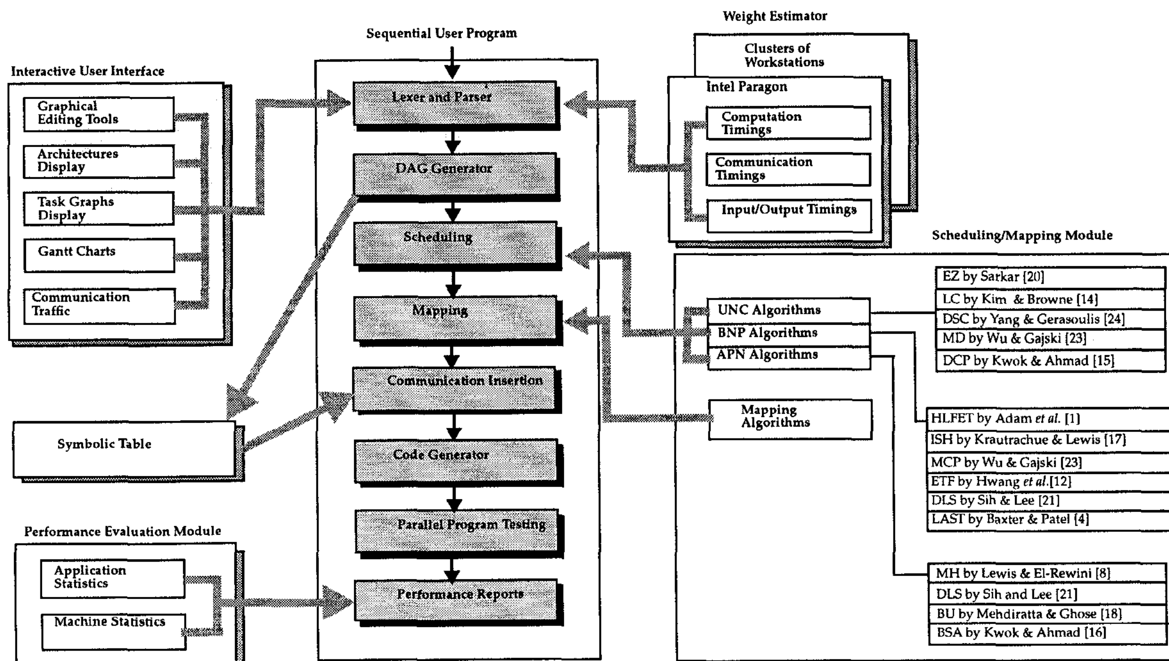

Figure 1: The system organization of CASCH.

scheduling technique [1], are suitable for scheduling when only a limited number of processors are available. The APN scheduling algorithms, which take into consideration link contention and the topology of target processor network, are useful for scheduling a distributed system.

Communication Insertion and Code Generation: Synchronization among the tasks running on multiple processors is carried out by communication primitives. The basic communication primitives for exchanging messages between processors are *send* and *receive*. They must be used properly to ensure a correct sequence of computation. These primitives are inserted automatically, reducing a programmer's burden and eliminating insertion errors. The procedure for inserting communication primitive is as follows. After scheduling and mapping, each node in a macro dataflow graph is allocated to a processor. If an edge leaves from a node to another node which belongs to a different processor, the *send* primitive is inserted after the node. Similarly, if an edge comes from another node in a different processor, the *receive* primitive is inserted before the node. However, if a message has already been sent to a particular processor, the same message does not need to be sent to the same processor again. If a message is to be sent to many processors, *broadcasting* or *multicasting* can be applied instead of separate message. After the communication primitives are properly inserted, parallel code is generated by including appropriate library procedures from a standard package such as the NX of the Intel Paragon.

3 Performance Results

CASCH runs on a SUN workstation that is linked through a network to an Intel Paragon. We have parallelized several applications on CASCH by using the scheduling algorithms described above (see Figure 1). In this paper we discuss the performance of two applications: Gaussian elimination and Laplace equation solver. The objective of including these results is to demonstrate the viability and usefulness of CASCH as well as to make a comparison among various scheduling algorithms. For reference, we have also included the results obtained with manually generated code. A manual code is generated by first partitioning the data among processors in a fashion that reduces the dependencies among these partitions. Based on this partitioning, an SPMD-based code is generated for each processors.

The performance measures include the program execution time (the maximum finish time out of all processors) measured on the Intel Paragon, the number of processors used by the schedule (and hence by the application program) generated by the scheduling algorithm, and the running time of the scheduling algorithm.

The first set of results (see Figure 2) are for the Gaussian elimination application with four different sizes of input matrix dimensions: 4, 8, 16, and 32. Figure 2(a) shows the execution times for various data sizes using different algorithms. We observe that the execution times vary considerably with different algorithms. Among the UNC algorithm, the DCP algorithm yields the best performance due to its superior scheduling method. Among the BNP algorithms, MCP and DLS are in general better, primarily because of their better task priority assignment methods. Among the APN algorithms, BSA and MH perform better,

Algorithm	Matrix Dimension 4	8	16	32
Manual	0.14	0.42	2.42	4.52
DCP	0.10	0.11	0.28	1.21
DSC	0.11	0.14	N.A.	N.A.
EZ	0.12	0.15	0.32	1.38
LC	0.10	0.13	0.32	1.42
MD	0.11	0.13	0.30	1.29
ETF	0.11	0.13	0.31	1.34
HLFET	0.11	0.14	0.35	2.47
ISH	0.11	0.14	0.34	1.29
LAST	0.12	0.16	0.33	1.50
MCP	0.11	0.16	0.30	1.28
DLS	0.11	0.14	0.29	1.30
BSA	0.11	0.13	0.32	1.42
BU	0.11	0.34	0.35	1.52
DLS	0.11	0.23	0.34	1.39
MH	0.12	0.26	0.33	1.28

(a) Execution Times (sec.) on the Paragon.

Algorithm	Matrix Dimension 4	8	16	32
Manual	4	8	16	32
DCP	3	7	10	22
DSC	5	22	95	128
EZ	1	2	15	33
LC	8	16	32	64
MD	2	3	4	7
ETF	3	7	16	32
HLFET	3	7	16	32
ISH	2	9	21	56
LAST	1	5	13	29
MCP	3	7	16	32
DLS	3	7	16	32
BSA	3	6	12	32
BU	8	20	38	48
DLS	3	7	12	16
MH	1	2	6	17

(b) Number of processors used.

Algorithm	Matrix Dimension (No. of Tasks) 4 (20)	8 (54)	16 (170)	32 (594)
DCP	6.22	6.48	13.29	281.54
DSC	0.04	0.05	0.09	0.23
EZ	6.19	6.50	14.81	330.15
LC	0.05	0.06	0.08	0.27
MD	6.21	6.84	38.91	2671.4
ETF	0.05	0.07	0.21	2.26
HLFET	0.06	0.07	0.15	0.69
ISH	0.05	0.07	0.08	0.32
LAST	0.05	0.08	0.24	2.23
MCP	0.05	0.07	0.09	0.40
DLS	0.04	0.08	0.28	2.84
BSA	0.74	1.96	13.57	224.05
BU	0.36	0.37	0.43	0.97
DLS	1.74	15.33	301.66	7459.3
MH	0.75	1.38	10.14	364.75

(c) Scheduling times (sec.) on a SPARC Station 2.

Figure 2: Execution times, number of processors used and scheduling times for the Gaussian elimination application.

due to their proper allocations of tasks and messages. All algorithms perform better than manually generated code: Compared to the manual scheduling, the level of performance improvement is up to 400%. The number of processors used by these algorithms is shown in Figure 2(b). The BU algorithm has a tendency of using a large number of processors. The times taken by various scheduling algorithms for generating the schedules for the Gaussian elimination example are included in Figure 2(c). We notice that these scheduling times vary drastically. The MD and DLS algorithms take considerably longer time to generate solutions while DSC and MCP are much faster.

Our second application is a Gauss-Seidel based algorithm to solve Laplace equations. The 4 matrix sizes used are 4, 8, 16, and 32. The application execution times using various algorithms and data size are shown in Figure 3(a). Again, using the best algorithms, such as DCP, more than 400% improvement over manually generated code is obtained. The UNC algorithms in general yield better schedules (mainly because they tend to use large numbers of processors). The numbers of processors used by these algorithms are shown in Figure 3(b). Again, the number of processors used by the DSC algorithm is quite large as compared to the other algorithms. The running times of the scheduling algorithms are shown in Figure 3(c) which are consistent with our earlier observations.

A number of conclusions can be made from the above results. First, in general UNC algorithms generate shorter schedules but uses more processors than BNP and APN

Algorithm	Matrix Dimension 4	8	16	32
Manual	0.72	2.89	24.12	72.32
DCP	0.72	1.06	6.08	16.02
DSC	0.72	1.34	6.30	16.42
EZ	0.72	1.44	6.95	18.28
LC	0.54	1.25	6.95	18.81
MD	0.72	1.25	6.52	17.08
ETF	0.72	1.25	6.73	17.75
HLFET	0.72	1.34	7.60	32.71
ISH	0.72	1.34	7.39	17.08
LAST	0.72	1.54	7.17	19.87
MCP	0.72	1.54	6.52	16.95
DLS	0.72	1.34	6.30	17.22
BSA	0.72	1.25	6.95	18.81
BU	0.59	3.26	7.60	20.13
DLS	0.72	1.92	8.69	23.25
MH	0.72	1.92	8.69	23.25

(a) Execution Times (sec.) on the Paragon.

Algorithm	Matrix Dimension 4	8	16	32
Manual	2	4	8	16
DCP	1	4	4	8
DSC	1	7	4	14
EZ	1	8	2	39
LC	2	4	4	8
MD	1	4	3	6
ETF	1	4	4	6
HLFET	1	4	4	7
ISH	1	5	4	14
LAST	1	3	2	5
MCP	1	4	4	7
DLS	1	4	4	6
BSA	1	5	4	7
BU	2	5	5	5
DLS	1	1	1	1
MH	1	1	1	1

(b) Number of processors used.

Algorithm	Matrix Dimension (No. of Tasks) 4 (18)	8 (66)	16 (258)	32 (1026)
DCP	8.58	7.03	9.60	44.95
DSC	0.07	0.04	0.10	0.29
EZ	8.63	7.15	9.71	35.00
LC	0.06	0.07	0.09	0.16
MD	8.58	7.65	10.01	111.99
ETF	0.05	0.07	0.10	0.30
HLFET	0.08	0.09	0.11	0.29
ISH	0.04	0.08	0.09	0.35
LAST	0.08	0.10	0.15	0.82
MCP	0.03	0.04	0.10	0.19
DLS	0.05	0.07	0.10	0.36
BSA	1.07	2.81	4.92	27.99
BU	0.47	0.37	0.50	0.73
DLS	1.66	7.81	10.86	75.90
MH	1.00	2.41	3.25	16.29

(c) Scheduling times (sec.) on a SPARC Station 2.

Figure 3: Execution times, number of processors used and scheduling times for the Laplace equation solver application.

algorithms. Thus, UNC algorithms are more suitable for MPPs. Second, BNP algorithms require less time for scheduling than UNC and APN algorithms and therefore are more suitable for scheduling under time constraint. Finally, APN algorithms tend to use less processors, due to its consideration of link contention, but generate slightly longer schedules for the Intel Paragon which has a fast network. Thus, APN algorithms are more suitable for distributed systems such as a network of workstations (NOW).

4 Conclusions and Future Work

The main objectives of CASCH are automatic parallelization and scheduling of applications to parallel processors. CASCH achieves these objectives by providing a unified environment for various existing and conceptual machines. Users can optimize their code by choosing the best algorithm. We are currently working on extending the capabilities of CASCH by including the following:

- including support for distributed computing systems such as a collection of diverse machines working as a distributed heterogeneous supercomputer system;
- extending the current database of benchmark timings by including more detailed and lower level timings of various computation, communication and I/O operations of various existing machines;
- including debugging facilities for error detection and global variable checking, etc.;
- expressing various kinds of parallelism, use a functional or logic programming language or object oriented language such as C++;
- designing a partitioning module for automatic or interactive partitioning of programs.

References

[1] T.L. Adam, K.M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. of the ACM*, vol. 17, pp. 685-690, Dec. 1974.

[2] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation and Comparison of Algorithms for Scheduling Task Graphs to Parallel Processors," Proc. of *the 1996 Int'l Symposium on Parallel Architecture, Algorithms and Networks,* Beijing, China, Jun. 1996, pp. 207-213.

[3] B. Appelbe and K. Smith, "A Parallel-Programming Toolkit," *IEEE Software*, pp. 29-38, Jul. 1989.

[4] J. Baxter and J.H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," Proc. *ICPP*, vol. II, pp. 217-222, Aug. 1989.

[5] Cray Research Inc., *UNICOS Performance Utilities Reference Manual*, sr2040 6.0 edition, 1991.

[6] Digital Equipment Corp., *PARASPHERE User Guide*.

[7] J.J. Dongarra and D.C. Sorensen, *Schedule Users Guide*, Tech. Rep. Version 1.1, Argonne National Lab., Jun. 1987.

[8] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *J. of Parallel and Dist. Computing*, vol. 9, no. 2, pp. 138-153, Jun. 1990.

[9] B.C. Gorda and E.D. Brooks III., "Gang Scheduling a Parallel machine," Technical Report UCRL--JC--107020, Lawrence Livermore National Laboratory.

[10] High Performance Fortran Forum, *High performance fortran language specification*, Technical Report Version 1.0, Rice University, May 1993.

[11] M.T. Heath and J.A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, 8(5):29-39, 1991.

[12] J.J.Hwang, Y.C. Chow, F.D. Anger and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing,* vol. 18, no. 2, pp. 244-257, Apr. 1989.

[13] K. Kennedy, K.S. McKinley, and C. Tseng, "Interactive Parallel Programming Using the Parascope Editor," *IEEE Trans. Parallel and Dist. Systems*, 2(3):329--341, 1991.

[14] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," Proc. *ICPP*, vol. II, pp. 1-8, Aug. 1988.

[15] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 5, May 1996, pp. 506-521.

[16] —, "Bubble Scheduling: A Quasi Dynamic Algorithm for Static Allocation of Tasks to Parallel Architectures," Proceedings of the *7th IEEE Symposium on Parallel and Distributed Processing,* Oct. 1995, pp. 36-43.

[17] T.G. Lewis and H. El-Rewini, "Parallax: A Tool for Parallel Program Scheduling," *IEEE Parallel & Distributed Technology*, vol.1, no. 3, pp. 62-72, May 1993.

[18] N. Mehdiratta and K. Ghose, "A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor," Proc. *ICPP*, vol. II, pp. 151-154, Aug. 1994.

[19] D. Pease, A. Ghafoor, I. Ahmad, K. Foudil-Bey, D. Andrews, T. Karpinski, M. Mikki and M. Zerrouki, "PAWS: A performance Assessment Tool for Parallel Computing Systems," *IEEE Computer*, vol. 24, no. 1, pp. 18-29, Jan. 1991.

[20] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.

[21] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-87, Feb. 1993.

[22] M. Wolfe, "The Tiny Loop Restructuring Research Tool," Proc. *ICPP*, vol. II, pp. 46-53, Aug. 1991.

[23] M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, 1(3):330-343, Jul. 1990.

[24] T. Yang and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors," *The 6th ACM Int'l Conf. on Supercomputing*, Jul. 1992.