A Graphical Tool for Automatic Parallelization and Scheduling of Programs on Multiprocessors†

Yu-Kwong Kwok[1], Ishfaq Ahmad[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. In this paper, we introduce an experimental software tool called CASCH (Computer Aided SCHeduling) for automatic parallelization and scheduling of applications to parallel processors. CASCH transforms a sequential program to a parallel program through automatic task graph generation, scheduling, mapping, communication, and synchronization primitives insertion. The major strength of CASCH is its extensive library of state-of-the-art scheduling and mapping algorithms reported in the recent literature. Using these algorithms, a practitioner can choose the most suitable one for generating the shortest schedule for the application at hand. Furthermore, the scheduling algorithms can be interactively analyzed, tested and compared using real data on a common platform with various performance objectives. CASCH with its graphical interface 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

The lack of automated parallel programming tools is one of the major problems faced by the programmers of parallel computers. The burden of coding a parallel algorithm is almost entirely on the programmer who must extract parallelism by deciding as to how to partition the data as well as control and coordinate inter-processor communication. An appropriate software development environment should allow an average programmer to write a parallel program without paying much attention to the characteristics of the underlying hardware. Even when parallelism is extractable in a compute-intensive application, its parallel code in the absence of efficient mapping and scheduling tools is not optimized and thus fails to yield a meaningful speedup. This problem is particularly serious for applications with irregular structures. Numerous such applications from the areas of digital signal processing, fluid dynamics, solution of linear and partial differential equations, and computer vision, etc., can run much faster if their codes are optimized through proper scheduling and mapping.

There are many experimental software tools for computer-aided parallel programming reported in the literature. Some of these tools are commercial products which are essentially parallel programs debugger [6], [7], [14], [15], [22], [29]. Some other tools, which are research prototypes, provide a more integrated program development environment which consists of program tracing and performance tuning facilities [3], [5], [9], [10], [12], [24], [25]. There have been a few advanced tools which include program transformation and restructuring facilities [16], [20], [21], [27], [30], [31], [32]. However, most of these advanced tools lack an easy-to-use user interface for designing and testing parallel applications interactively. Furthermore, these tools have only a limited applicability as they do not provide an extensive library of scheduling and mapping tools suitable for different environments.

In this paper, we describe a graphical software tool called CASCH (Computer Aided

†. 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.

SCHeduling) for parallel processing on distributed-memory multiprocessors. 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 can be considered to be a super set of tools such as PAWS [24], Hypertool [31], PYRROS [32], and Parallax [20], since it includes the major functionalities of these tools at a more advanced and comprehensive level and also offers additional useful features.

CASCH is a unique tool in that it provides all of the important ingredients for developing parallel programs. It frees the user from carrying out several tedious chores and can significantly improve the performance of a parallel program. It is useful for a novice parallel programmer due to its automatic parallelization and code generation facilities. It can also help an experienced researcher since it provides various facilities to fine-tune and optimize a parallel program. CASCH includes from the recent literature an extensive library of state-of-the-art scheduling algorithms which are organized into different categories suiting different architectural environments. These 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 inserted using a database that contains the benchmark timing of various computation, communication, and I/O operations for different machines. An attractive feature of CASCH is its graphical user interface which provides a flexible and easy-to-use interactive environment for analyzing various scheduling and mapping algorithms, using task graphs generated randomly, interactively, or directly from real programs. Multiple windows can be opened to show the schedules of task graphs generated by different scheduling algorithms for a given machine.

This paper is organized as follows. Section 2 gives an overview of CASCH and describes it major functionalities. We briefly introduce the graphical user interface of CASCH in Section 3. Section 4 includes some preliminary results of the experiments conducted on the Intel Paragon using CASCH. We provide some concluding remarks in the last section.

2 Overview of CASCH

The system organization of CASCH is shown in Figure 1. Using the CASCH tool, the user first writes a sequential program from which a DAG 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.

Figure 2(a) shows an example, a sequential fast Fourier transform algorithm, in which the data matrix is partitioned by columns across processors. The procedures *InterMult* and *IntraMult* are called several times. 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.

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.

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 have been obtained through benchmarking using

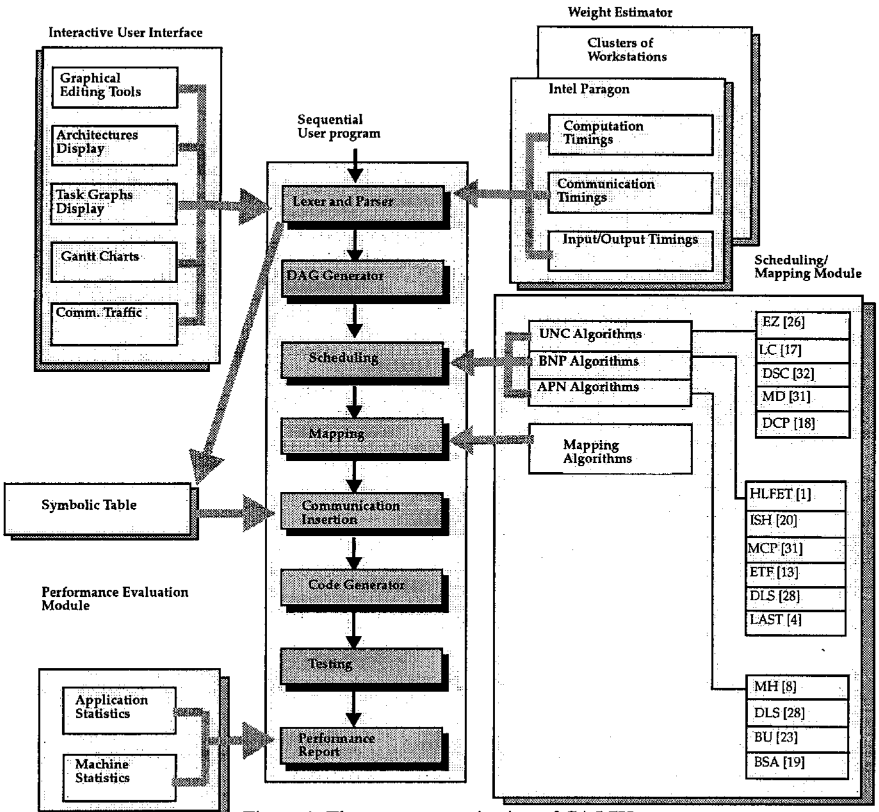


Figure 1: The system organization of CASCH.

an approach similar to [24]. Communication estimation, which is obtained experimentally, is based on the cost for each communication primitive, such as *send*, *receive*, and *broadcast*. The current version of the computation estimator is a symbolic estimator. The estimation is based on reading through the code without running the code. Its symbolic output is in the form of a function of input parameters of a code.

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 in the graph 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 PE (processing element) in the target machine, the weight of the edge connecting them becomes zero. In static scheduling, the number of nodes is known before program execution. 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.

A common approach to distribute workload to processors is partitioning a problem into P tasks and performing a one-to-one mapping between the tasks and the processors. Partitioning can be done with the "block", "cyclic", or "block-cyclic" pattern [11]. 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

```
Program FFT

    /* N: number of points for discrete Fourier Transform, let N=PNxSN  */
    /* data[log(PN)+2][PN][SN]                                           */
    /*    stores single-assigned data points for discrete Fourier        */
    /*    Transform organized as a PN x SN grid for parallel computation */
    /*------------------------- main program --------------------------*/

    call Initiation;        /* serial part; initialize the array `data' */

        /* parallel inter-multiplication of data points                  */
    for i = log(PN) downto 1 do
        for j = 0 to PN-1 step 1<<i do
            for k = 0 to 1<<(i-1)-1 do
                call InterMult(data[i+1][j+k], data[i+1][j+k+1<<(i-1)],
                               data[i][j+k], SN);
                call InterMult(data[i+1][j+k+1<<(i-1)], data[i+1][j+k],
                               data[i][j+k+1<<(i-1)], SN);
                    /* in each iteration, InterMult can be executed if  */
                    /* arrays data[i+1][j+k] and data[i+1][j+k+1<<(i-1)]*/
                    /* are available upon completion, data[i][j+k] and  */
                    /* data[i][j+k+1<<(i-1)] will be avaliable          */
            endfor
        endfor
    endfor

        /* parallel intra-multiplication of data points                  */
    for i = 0 to PN-1 do
        call IntraMult(data[1][i], data[0][i], SN);
            /* in each iteration, IntraMult can be executed if array    */
            /* data[1][i] is avaliable; upon completion, data[0][i],    */
            /* which is the result, will be avaliable                   */
    endfor

    call OutputResult;      /* serial part; inverse and return results  */
EndProgram FFT

/*------------------------- Procedure InterMult -----------------------*/
Procedure InterMult(inArray1, inArray2, outArray, n)

    /* Input:   inArray1, inArray2   data points for multiplication    */
    /*          n                    number of data points in sub-array */
    /* Output:  outArray             array of output data              */

    for i = 0 to n-1 do
        outArray[i] = inArray1[i] @ inArray2[i];/* '@' is element-wide  */
                                               /* complex FFT operation*/
    endfor
 EndProcedure InterMult
/*------------------------- Procedure IntraMult -----------------------*/
Procedure IntraMult(inArray, outArray, n)

    /* Input:   inArray              data points for multiplication    */
    /*          n                    number of data points in sub-array */
    /* Output:  outArray             array of output data              */

    for i = log(n) downto 1 do
       for j = 0 to n-1 step 1<<i do
          for k = 0 to 1<<(i-1)-1 do
             outArray[j+k] = inArray[j+k] @ inArray[j+k+1<<(i-1)];
             outArray[j+k+1<<(i-1)] = inArray[j+k+1<<(i-1)] @ inArray[j+k];
             /* where '@' is element-wide complex FFT operation        */
          endfor
       endfor
       for j = 0 to n-1 do
          inArray[j] = outArray[j];
       endfor
    endfor
EndProcedure IntraMult
```

Figure 2(a): A sequential program for fast Fourier transform.

```
/* For PE 0      */
/* load array of data points from HOST */
receive(HOST, data[3][0]);
receive(HOST, data[3][1]);
receive(HOST, data[3][2]);
receive(HOST, data[3][3]);

InterMult(data[3][3],data[3][1],data[2][3],2);
send(PE1, data[2][3]);

InterMult(data[3][1],data[3][3],data[2][1],2);

InterMult(data[3][2],data[3][0],data[2][2],2);
send(PE1, data[2][2]);

InterMult(data[3][0],data[3][2],data[2][0],2);

InterMult(data[2][1],data[2][0],data[1][1],2);
send(PE2, data[1][1]);

InterMult(data[2][0],data[2][1],data[1][0],2);
send(PE2, data[1][0]);

InterMult(data[2][3],data[2][2],data[1][3],2);

IntraMult(data[1][3],data[0][3],2);

/* unload result array of data points to HOST */
send(HOST, data[0][3]);

/* For PE 1      */
receive(PE0, data[2][2]);
receive(PE0, data[2][3]);
InterMult(data[2][2],data[2][3],data[1][2],2);

IntraMult(data[1][2],data[0][2],2);

/* unload result array of data points to HOST */
send(HOST, data[0][2]);

/* For PE 2      */
receive(PE0, data[1][1]);
IntraMult(data[1][1],data[0][1],2);

receive(PE0, data[1][0]);
IntraMult(data[1][0],data[0][0],2);

/* unload result array of data points to HOST */
send(HOST, data[0][1]);
send(HOST, data[0][0]);
```

Figure 2(b): The parallel code for fast Fourier transform.

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. 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 reader is referred to [2] for a qualitative and quantitative comparison of different classes of scheduling algorithms. After applying different algorithms to the task graph, the user can choose the best one which generates the shortest schedule.

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 can be 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 has been allocated to a PE. If an edge leaves from a node to another node which belongs to a different PE, the *send* primitive is inserted after the node. Similarly, if an edge comes from another node in a different PE, the *receive* primitive is inserted before the node. However, if a message has already been sent to a particular PE, the same message does not need to be sent to the same PE again. If a message is to be sent to many PEs, *broadcasting* or *multicasting* can be applied instead of separate message.

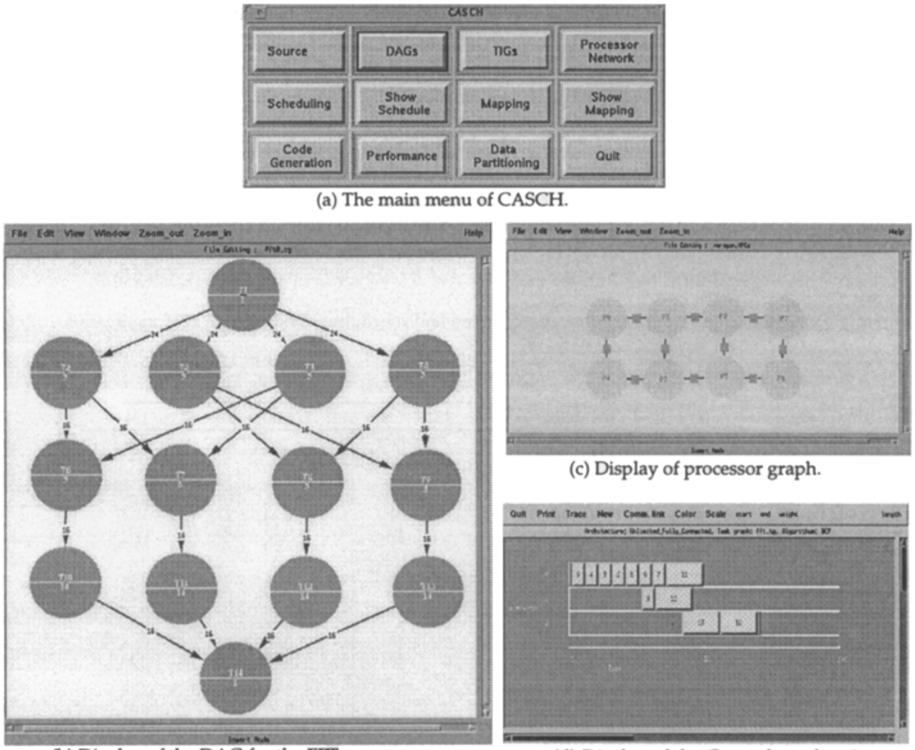
The insertion method described above does not ensure a correct communication sequence. Thus, we use a *send-first* strategy for a reordering of communication primitives. That is, we reorder *receive*s according to the order of *send*s. Reordering of *send*s and *receive*s may not be necessary for a system supporting typed messages. However, even for such systems, message transmission reordering may reduce the message waiting time and the demand for communication buffers.

We use the example of Figure 2(a) to illustrate our code generation method. Figure 2(b) shows the generated parallel code for three PEs (assuming N= 8). Note that only the main program for each PE is shown. The data structure is the same as in Figure 2(a). The initial data is loaded to PEs such that each PE obtains the portion of data required for its computation. Consequently, the memory space is compacted. In this example, the initial data is loaded to PE 0. To reduce the number of message transfers and, consequently, the time to initiate messages, several messages can be packed and sent together. For example, the first four messages can be packed into one message and sent to PE 0. Finally, the fourth data partition of the result is unloaded from PE 0, the third from PE 1, and the first and the second from PE 2.

3 Graphical User Interface

The graphical capabilities of CASCH provide the user with a an easy-to-use window-based interactive interface (the main menu of this interface is shown in Figure 3(a)). The graphical interface includes the facilities following:

- **Source:** The user can create, edit, or browse through sequential programs. The *source* button also includes a sub-menu for generating a DAG from the user program.
- **DAGs:** This includes facilities to display a DAG generated from the user program (Figure 3(b) shows the task graph for the FFT program). Other options include displaying a randomly generated DAG or creating a DAG interactively. The editing facilities include node and edge insertion; node IDs as well as weights on the node and edges can be labelled with numbers.
- **TIGs:** This facility is similar to *DAGs* except that edges in task graphs are not directed but are undirected, that is, task graphs are TIGs.
- **Processor Network:** This facility allows the user to display a processor architecture (including the processors and the network topology). The editing facilities, similar to *DAGs*, allow the user to interactively create various network topologies. An example processor graph is illustrated in Figure 3(c).
- **Scheduling:** This facility includes a sub-menu from which the user must first select one of the three classes of the scheduling algorithms, i.e., BNP, UNC, and APN. Within each class, the user needs to select one of the scheduling algorithms. The algorithm, for its execution, then requires the user to enter a number of parameters.
- **Show Schedule:** The schedule generated as the results of invoking a scheduling algorithm can be displayed using this facility (Figure 3(d) shows the schedule of the FFT example). The schedule is displayed using a Gantt chart showing the start and finish times of tasks on various processors. Clicking on any task in the Gantt chart displays its start and finish times; the total schedule length is shown in the right corner of the window. A schedule also includes communication messages on the network (displayed through another window which is invoked by clicking on any two processors). The scale of the display can be changed to zoom the Gantt chart. An important features of this facility is the *trace* option which shows a step-by-step scheduling of each task.
- **Code generation**: Clicking on this button simply generates a parallel code for a given program according to a schedule/map generated by a scheduling/mapping algorithm.



(a) The main menu of CASCH.

(b) Display of the DAG for the FFT program.

(c) Display of processor graph.

(d) Display of the Gantt chart showing the schedule for the FFT program.

Figure 3: The graphical interface of CASCH.

- **Performance:** The performance facilities include processors utilization, time spent in computation and communication, and speedup.
- **Data Partitioning:** This facility includes tools for displaying structured and unstructured meshes as well as partitioning of data across different processors.

4 Results

CASCH runs on a SUN workstation that is linked through a network to an Intel Paragon. We have parallelized several applications, including FFT, Gaussian elimination, Laplace equation solver, matrix multiplication and N-body problem, on CASCH by using all of the scheduling algorithms described above. Due to space limitations, we describe the results for the FFT application only. 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 target processors used and the running time of the scheduling algorithm.

The results shown in Table 1 are for the FFT example discussed earlier with four different sizes of input data: 8, 32, 128, and 512 points. The first four columns of Table 1 show 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, 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 numbers of processors used by the algorithms are shown in the middle four columns of Table 1. The times taken by various scheduling algorithms for generating the schedules for the FFT example are included in the last four columns of Table 1. As can be seen, these times vary drastically. The MD and DLS algorithms take considerably longer time to generate solutions (due to their dynamic calculation of priorities) while DSC and MCP are much faster (due to their simple priority calculation mechanism).

Table 1: Execution times, number of processors used and scheduling times for the FFT application.

	Execution Times (sec.) on the Intel paragon				*Number of processors used*				*Scheduling times (sec.) on a SPARC Station 2.*			
	Number of Points				Number of Points				Number of Tasks			
Algorithm	8	32	128	512	8	32	128	512	14	34	82	194
Manual	0.07	0.47	1.76	2.65	3	6	9	12	—	—	—	—
DCP	0.06	0.11	0.63	0.64	3	5	10	18	6.24	6.26	7.06	15.80
DSC	0.06	0.15	0.66	0.71	3	7	12	30	0.08	0.04	0.06	0.09
EZ	0.07	0.16	0.72	0.78	3	13	34	64	6.22	6.27	7.00	15.37
LC	0.06	0.14	0.72	0.81	4	8	16	32	0.05	0.05	0.05	0.09
MD	0.06	0.14	0.68	0.73	3	4	12	21	6.20	6.30	8.74	69.78
ETF	0.07	0.14	0.69	0.76	3	6	10	14	0.04	0.06	0.07	0.16
HLFET	0.07	0.15	0.79	1.40	3	6	10	16	0.03	0.05	0.09	0.15
ISH	0.07	0.15	0.77	0.73	3	7	14	24	0.05	0.03	0.05	0.13
LAST	0.07	0.17	0.74	0.86	2	4	8	14	0.03	0.07	0.08	0.30
MCP	0.07	0.17	0.68	0.72	3	6	10	14	0.05	0.03	0.08	0.14
DLS	0.07	0.15	0.66	0.73	3	6	10	14	0.08	0.08	0.12	0.29
BSA	0.07	0.14	0.72	0.74	3	5	5	5	0.60	0.95	2.15	5.38
BU	0.06	0.36	0.79	0.86	10	22	37	26	0.34	0.35	0.37	0.41
DLS	0.07	0.24	0.77	0.79	3	5	5	5	1.00	3.79	18.20	93.35
MH	0.07	0.27	0.74	0.72	3	5	5	5	0.58	0.94	1.93	4.58

We can make a number of conclusions based on the above results.

- In general, the UNC algorithms generate shorter schedules but uses more processors than BNP and APN algorithms. Thus, UNC algorithms are more suitable for MPPs.
- The BNP algorithms require less time for scheduling than UNC and APN algorithms and therefore are more suitable for scheduling under time constraint.
- The 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).

5 Conclusions

The design objectives of CASCH are automatic parallelization and scheduling of applications to parallel processors so that performance of applications is optimized. CASCH achieves these objectives by providing a unified environment for various existing and conceptual machines. An attractive feature of CASCH is that it provides an easy-to-use graphical framework for users to compare various scheduling algorithms from its extensive library. Users can generate schedules using different scheduling algorithms and, by visualizing the quality of the schedules displayed in the graphical windows, choose the best one for final code generation.

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. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 217-222, Aug. 1989.

[5] A. Beguelin, "A tool for monitoring {PVM} programs," Tech. Rep. CMU-CS-93-105, CMU, 1993.

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

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

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

[9] C. Farhat and M. Lesoinne, "Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics," *International Journal for Numerical Methods in Engineering*, 36(5):745--764, 1993.

[10] C. Fineman, P. Hontalas, S. Listgarten, and J. Yen, *A User's Guide to AIMS*, ver. 1.1 ed., May 1992.

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

[12] M.T. Heath and J.A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, 8(5):29-39, 1991.

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

[14] Intel Supercomputer Systems Division, *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual*, Apr. 1991.

[15] Kendall Square Research, KSR Manual.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] V.M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M.A. Mohamed, B. Nitzberg, J.A. Telle, and X. Zhong, "OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures," *International Journal of Parallel Programming*, vol. 20, no. 3, 1991, pp. 237-270.

[22] MasPar Computer, MPPE User Guide.

[23] N. Mehdiratta and K. Ghose, "A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor," Proc. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 151-154, Aug. 1994.

[24] D. Pease, A. Ghafoor, I. Ahmad, K. Foudil-Bey, D. Andrews, T. Karpinski, M. Mikki and M. Zerrouki, "PAWS (Parallel Assessment Window System): A performance Assessment Tool for Parallel Computing Systems," *IEEE Computer*, Vol. 24, No. 1, pp. 18–29, January 1991.

[25] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shield, and B. W. Schwartz., "The Pablo Performance Analysis Environment," Technical report, Computer Science Department, University of Illinois at Urbana-Champagn.

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

[27] B. Shirazi, K. Kavi, A.R. Hurson, and P. Biswas, "PARSA: A Parallel Program Scheduling and Assessment Environment," Proceedings of *International Conference Parallel Processing*, 1993, vol. II, pp. 68-72.

[28] 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.

[29] Thinking Machines Corp., Prism User's Guide, version 1.1 edition, December 1991.

[30] M. Wolfe, "The Tiny Loop Restructuring Research Tool," *Int'l Conf. on Parallel Processing*, pages II--46--53, August 1991.

[31] 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.

[32] 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.