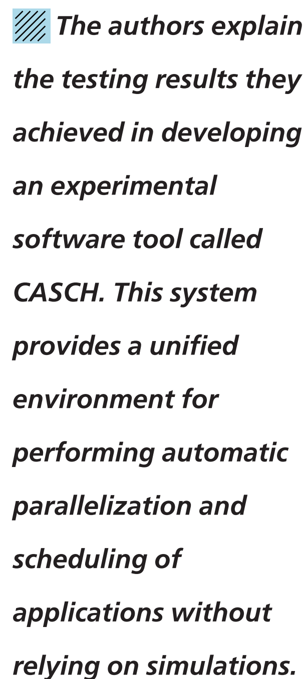
Ishfaq Ahmad
Hong Kong University of Science and Technology

Yu-Kwong Kwok
University of Hong Kong

Min-You Wu, and Wei Shu
University of Central Florida



The authors explain the testing results they achieved in developing an experimental software tool called CASCH. This system provides a unified environment for performing automatic parallelization and scheduling of applications without relying on simulations.

CASCH: A Tool for Computer-Aided Scheduling

Programmers of parallel computers regard automated parallel-programming environments as highly desirable. Software tools embedded in a parallel-programming environment can carry out numerous tasks, such as interprocessor communication and proper scheduling. They free the average programmer from the major hurdles of parallelization—potentially improving their performance—and, because manually performing these tasks can be tedious, they also help experienced programmers.

Even though a large body of literature exists in the area of scheduling and mapping [1–3] (see the "Recent research" sidebar), people have exploited only a part of it for practical purposes. While some have proposed software tools that support automatic scheduling and mapping, those tools' main function is to provide a simulation environment.[4] They can help us understand how scheduling and mapping algorithms operate and behave, but they are inadequate for practical purposes. On the other end of the spectrum, there are numerous parallelizing tools (see the "Parallel programming tools" sidebar), but they are usually not well integrated with sophisticated scheduling algorithms.

We have designed a software tool called CASCH (*C*omputer-*A*ided *Sch*eduling) for parallel processing on distributed-memory multiprocessors in a complete parallel programming environment, including parallelization, partitioning, scheduling, mapping, communication, synchronization, code generation, and performance evaluation. A compiler automatically converts sequential applications into parallel codes to perform program parallelization. The parallel code that executes on a target machine is optimized by Casch through proper scheduling and mapping.

Overview of CASCH

CASCH is unique because it provides an integrated programming environment for developing parallel programs. Its automatic parallelization and code generation helps naïve users, and it provides experienced programmers with various facilities to fine-tune and optimize a program. It also includes an extensive library of state-of-the-art scheduling algorithms described in recent literature,[4] organized into different categories that are suitable for different architectural environments. The user can select one of these algorithms for scheduling the task graph the application

1092-3063/00/$10.00 © 2000 IEEE

Recent research

The most common models of a parallel program are the precedence-constrained directed acyclic graph (DAG) and the task interacting graph (TIG) with no temporal dependencies. Figure A1 shows a parallel loop nest, and Figure A2 depicts the DAG representing the loop.

The weight associated with a task represents the amount of execution time of the corresponding task, and the weight associated with an edge represents the amount of communication time. Numerous techniques have been proposed for generating the task and edge weights offline such as execution profiling and analytical benchmarking.[1] With such a *static* model, we invoke a scheduler offline during compile-time. We call this form of the multiprocessor-scheduling problem *static scheduling* or DAG scheduling.

Figure B provides a taxonomy of static parallel scheduling algorithms. The taxonomy is partial because it does not include details of some of the earlier work on scheduling. We considered only those scheduling algorithms that we can use in a realistic environment and that are relevant in our context. The taxonomy is hierarchical and develops by expanding each layer. Thick arrows indicate the relevance to our discussion and a further division of a particular layer; the thin arrows do not lead to a further division in the taxonomy.

The highest level of the taxonomy is divided into two categories, depending on whether the tasks are independent. We limit our discussion to dependent tasks. Earlier algorithms make simplifying assumptions about the task graph representing the program and the model of the multiprocessor system. Some algorithms ignore the precedence constraints and consider the task graph to be free of temporal dependencies (task interacting graph). The algorithms considering the more realistic task precedence–constrained graph assume the graph to be of a special structure such as tree, forks-join, and so forth. In general, however, parallel programs come in a variety of structures. We can divide the algorithms designed to tackle arbitrary graph structures into two categories. Some algorithms assume the computational costs of all the tasks to be the same; others assume the computational costs of tasks to be arbitrary. It is worth mentioning that the scheduling problem is NP-complete even in two simple cases: scheduling unit-time tasks to an arbitrary number of processors and scheduling one or two time unit tasks to two processors.

We may perform scheduling with communication with or without duplication of tasks.[2] Each class can further subdivide into two categories. Note that only the division of *No-Duplication* class is shown. An exact division of *Duplication* can also be envisaged but is not shown here due to its similarity with the *No-Duplication* class.

Some scheduling algorithms assume the availability of an unlimited number of processors [3–7] with a fully connected network. These are called the UNC (unbounded number of clusters) scheduling algorithms. The algorithms assuming a limited number of processors are called the BNP (bounded number of processors) scheduling algorithms. In the UNC and BNP scheduling algorithms, we assume the processors are fully connected, and we ignore link contention or routing strategies used for communication. If scheduling and mapping are done in separate steps, the schedules the UNC or BNP algorithms generate can be mapped onto the processors using the indirect mapping approach. The algorithms that assume the system to have an arbitrary network topology are called the APN (arbitrary processor network) scheduling algorithms.[8]

The basis of a major component of scheduling algorithms (in all three classes) is the classical list–scheduling approach.[9,10] In list scheduling, the scheduler assigns the tasks priorities and places the tasks in a ready list arranged in a descending order of priority. The task with a higher priority is examined for scheduling before a task with a lower priority. If more than one task has the same priority, ties are broken using some method. A task selected for scheduling is allocated to a processor that allows the earliest start time. After a task is scheduled, more tasks may be added in the ready list. Again, the tasks in the ready list are examined and scheduled. This continues until all tasks are scheduled.

The scheduling algorithm library of CASCH includes five UNC, six BNP, and four APN algorithms. The major characteristics of these algorithms are briefly described in the main text. See Ahmad, Kwok and Wu[3] for a more detailed description and comparison.

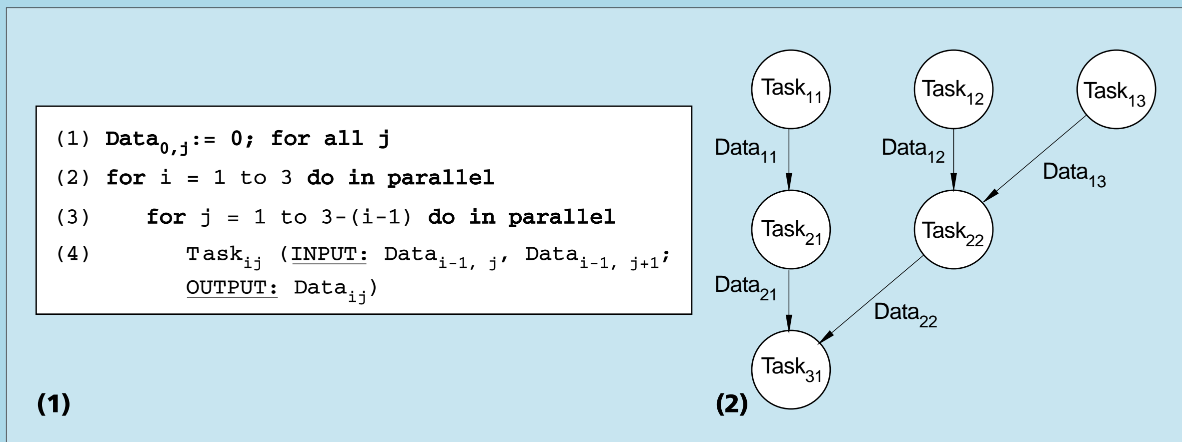


Figure A. (1) A parallel program fragment and (2) a directed acyclic graph representing the program fragment.

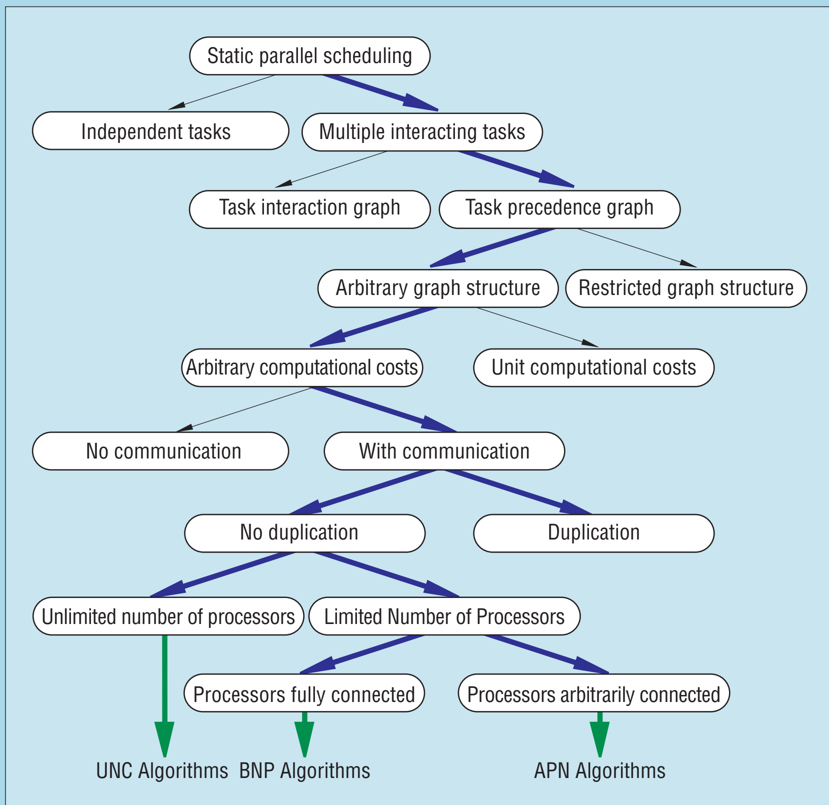


Figure B. A partial taxonomy of the multiprocessor-scheduling problem.

References

1. K. Hwang, Z. Xu, and M. Arakawa, "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing," *IEEE Trans. Parallel and Distributed Systems,* Vol. 7, No. 5, May 1996, pp. 522–536.
2. I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems*, Vol. 9, No. 9, Sept. 1998, pp. 872–892.
3. I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," *Proc. 2nd Int'l Symposium on Parallel Architecture, Algorithms, and Networks*, June 1996, pp. 207–213
4. S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing*, Vol. II, Aug. 1988, pp. 1–8.
5. Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors ," *ACM Computing Surveys*, Vol. 31, No. 4, Dec. 1999, pp. 406–471.
6. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, Mass., 1989.
7. T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 5, No. 9, Sept. 1994, pp. 951–967.
8. G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp. 75–78.
9. T.L. Adam, K.M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. ACM,* Vol. 17, No. 12, Dec. 1974, pp. 685–690.
10. H. El-Rewini and T.G. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *J. Parallel and Distributed Computing*, Vol. 9, No. 2, June 1990, pp. 138–153.

generates. The weights on the tasks and edges of the task graph are inserted using a database. It contains the timing of various computation, communication, and I/O operations for different machines, obtained through benchmarking.

Figure 1 shows CASCH's overall organization; the main components are its

- compiler, which includes a lexical analyzer and a parser,
- task graph generator,
- weight estimator,
- scheduling and mapping module,
- communication inserter,
- code generator,
- interactive user interface, and
- program testing and performance evaluation module.

Using CASCH, the user first writes a sequential program, which generates a task graph. To automate program development, the sequential program is composed of a set of procedures invoked by the main program. Using the single assignment rule, the programmer should write each procedure as an indivisible unit of computation to be scheduled by CASCH. The programmer then determines the procedures' grain sizes and can modify them.

Figure 2 shows an example (an implementation of a Fast Fourier Transform algorithm) in which columns across processors partition the data matrix. In the serial program, the constant $N = PN \times SN$ determines the problem size. Specifically, the constants *PN* and *SN* control the granularity of the partitioning. The larger the value of *SN*, the higher the granularity. In the current implementation of CASCH, the user defines these constants at compile-time. The procedures *InterMult* and *IntraMult* are invoked by the main program several times. The user can ignore the control dependencies so he or she can assume a procedure executes whenever all input data of the procedure are available. The single assignment of parameters in procedure calls defines data dependencies. The user can invoke communications only at the beginning and

the end of procedures. In other words, a procedure receives messages before it begins execution and sends messages after it has finished execution.

In general, the user must implement the application (for example, an FFT) only in the form of a sequential program consisting of a set of procedures. The sequential program is basically an ordinary C program except that the user must insert a few annotations in the form of *#define* compiler directives that instruct CASCH to invoke data-array partitioning. For instance, in the FFT example, the user just needs to define values of *PN* and *SN* in the header of the sequential C program. In this example, $PN = 4$ and $SN = 2$.

Lexical analyzer and parser

The lexical analyzer and parser analyze the data dependencies and user defined partitions. In our implementation of CASCH, we constructed both components with the help of *lex* and *yacc*. If CASCH discovers a syntax or semantic error in this stage, it advises the user to fix the problem before proceeding to the task-graph-generation phase. For a static program, the user knows the number of procedures before program execution. Many numerical types of applications belong to this static class of programs.[5] This type of program is system independent because the program does not specify communication primitives. Data dependencies among the procedural parameters define a *macro dataflow graph* (that is, the task graph).[6]

Task graph generation

The main program generates a macro dataflow graph—a directed acyclic graph (DAG) with start and end points. A macro dataflow graph consists of a set of tasks $\{T_1, T_2, \ldots, T_n\}$ and a set of edges $\{e_1, e_2, \ldots, e_m\}$ such that $e_k = T_i \rightarrow T_j$ for $1 \le k \le m$ and some i, j, where $1 \le i, j \le n$. Each node in the graph corresponds to a procedure or a task, and the procedure execution time represents the task weight. Each edge corresponds to a message transferred from one procedure to

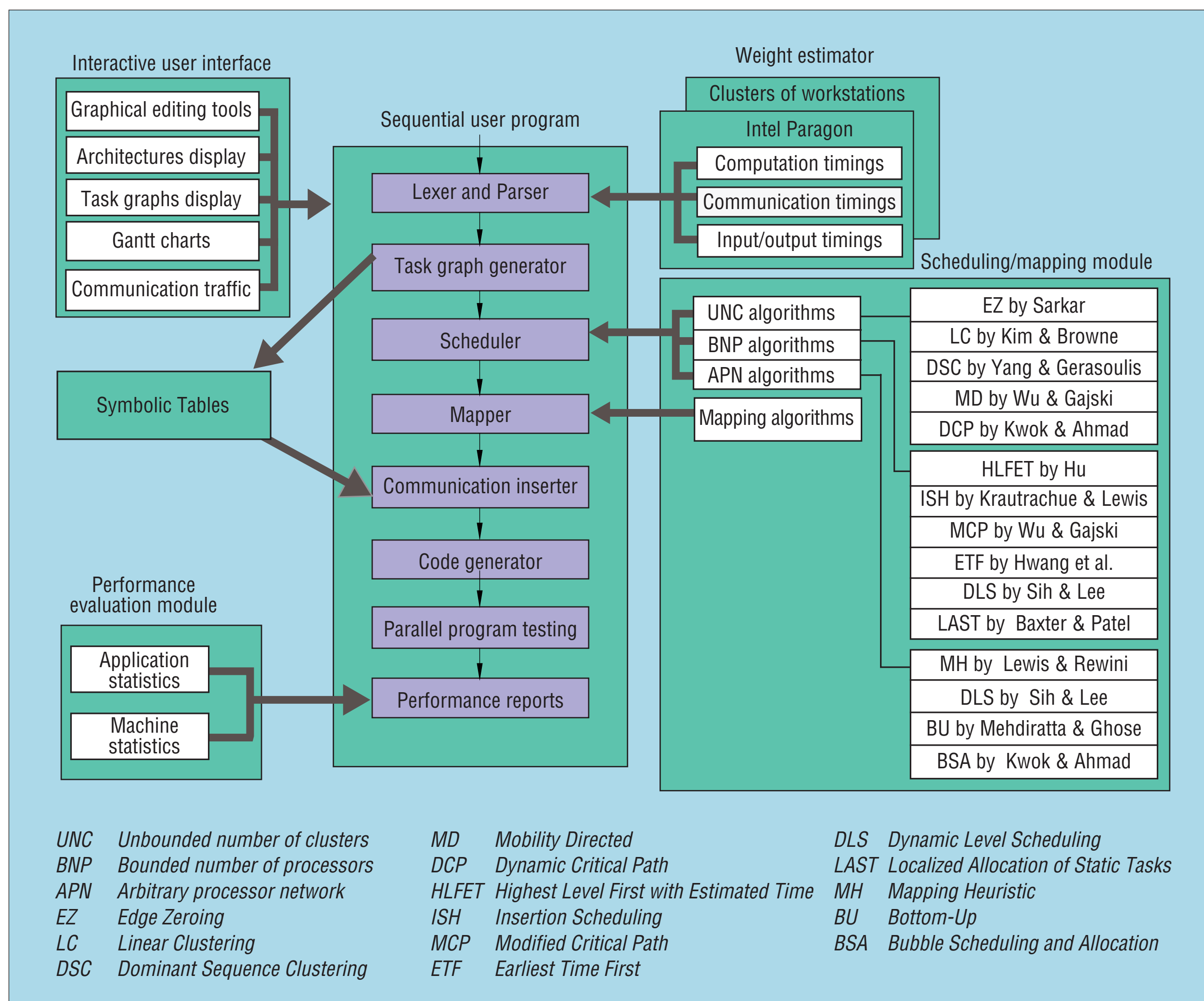


Figure 1. The various components and functionalities of Casch.

```
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 available */
                      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 available; upon completion, data[0][i], */
                      /* which is the result, will be available */
      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 sequential program for a Fast Fourier Transform.

Other Parallel Programming Tools

Several research efforts have demonstrated the usefulness of program development tools for parallel processing on message-passing multiprocessors. Essentially, these tools fall into two classes. The first class, which is mainly comprised of commercial tools, provides software development and debugging environments. The Atexpert by Cray Research[1] is an example. Some of these tools also provide performance-tuning tools and other program development facilities. The second class performs some program transformation through program restructuring. Parascope[2] and TINY[3] are restructuring tools that automatically transform sequential programs to parallel programs. TOP/DOMDEC[4] is a tool for program partitioning. Some of the recently reported prototype scheduling tools are described below.

PAWS is a performance evaluation tool that provides an interactive environment for performance evaluation of various multiprocessor systems.[5] PAWS does not perform scheduling and mapping and does not generate any code. It is useful only for simulating the execution of an application on various machines.

Hypertool takes a user-partitioned sequential program as input and automatically allocates and schedules the partitions to processors.[6] Proper synchronization primitives also are automatically inserted. Hypertool is a code generation tool since the user program is compiled into a parallel program for the iPSC/2 hypercube computer using parallel code synthesis and optimization techniques. The tool also generates performance estimates including execution time, communication and suspension times for each processor, as well as network delay for each communication channel. Scheduling is done using the MD algorithm or the MCP algorithm.

PYRROS is a compile-time scheduling and code generation tool.[7] Its input is a task graph and the associated sequential C code. The output is a static schedule and a parallel C code for iPSC/2. PYRROS consists of a task graph language with an interface to C. This scheduling system uses only the DSC algorithm, an X Windows-based graphic displayer, and a code generator. The task graph language allows the user to define partitioned programs and data. The scheduling system's job is to cluster the task graph, perform load-balanced mapping, and do computation/communication ordering. The graphic displayer displays task graphs and scheduling results in the form of Gantt charts. The code generator inserts synchronization primitives and performs parallel code optimization for the target parallel machine.

Parallax incorporates seven classical scheduling heuristics designed in the 1970s.[8] This provides an environment for parallel program developers to find out how the schedulers affect program performance on various parallel architectures. Users must provide the input program as a task graph and estimate task execution times. Users must also express the target machine as an interconnection topology graph. Parallax then generates schedules in the form of Gantt charts, speedup curves, processor, and communication efficiency charts using an X Windows interface. In addition, an animated display of the simulated running program helps developers to evaluate the differences among the provided scheduling heuristics. Parallex, however, is not reported to generate executable parallel code.

Oregami is designed for use in conjunction with parallel programming languages that support a communication model.[9] These models can be OCCAM, C*, or traditional pro-

Table 1. Communication timing constants (microseconds) for various target machines

Machine	Start-up	Rate per byte	1/Clock rate
Intel Paragon	150	0.40	0.02000
IBM SP-2	42	0.14	0.00625

another procedure. The edge's weight equals the message's transmission time. When the scheduler assigns two tasks to a single processor, the weight of the edge connecting them becomes zero.

Weight estimator

The task graph generator inserts the weights on the tasks and edges with the help of an estimator, which provides the cost (measured in time) of executing various instructions and the cost of communication on a given machine. We obtained these timings through benchmarking using an approach similar to analytical benchmarking and profiling.[8,9]

Communication estimation, obtained experimentally, is based on the cost for each communication primitive, such as *send*, *receive*, and *broadcast*. Our approach is similar to that used by Xu and Hwang.[7] Table 1 shows the communication times (assuming a stand-alone mode) for various target machines.

The current version of the computation estimator is a symbolic estimator. The estimation is based on reading through the code without running it. Its symbolic output is in the form of a function of the code's input parameters. With a symbolic estimator and a restricted class of C codes, the code does not need reestimation for different problem sizes. The code might include functions and procedures, and the estimator generates performance for each of them. The code might also have *for loops*. A loop's boundaries can be either constants or input parameters. The cost of each operation or built-in function is specified in the *cost files*. Summing all costs of operations and functions for a segment of code provides the total cost of the computation.

Task scheduling and mapping

A common approach to distribute the workload among p 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.[6] These partitioning schemes—using simple scheduling heuristics such as the "owner computes" rule—work for certain problems but could fail for many others. This is especially the case with irregular problems, because it is difficult to balance the load and minimize dependencies simultaneously. We handle an irregular problem by partitioning it into

gramming languages like C and Fortran extended with communication facilities. It is a set of tools that includes a LaRCS compiler to compile textual user task descriptions into specialized task graphs called TCG (Temporal Communication Graphs). Plus, Oregami includes a mapper tool for mapping tasks on a variety of target architectures and metric tools for analyzing and displaying the performance. The suite of tools is implemented in C for Sun workstations with an X Windows interface. However, precedence constraints among tasks are not considered in Oregami. Moreover, no target code is generated.

PARSA is a software tool developed for automatic scheduling and partitioning of sequential user programs.[10] PARSA consists of four components: an application specification tool, an architecture specification tool, a partitioning and scheduling tool, and a performance assessment tool. PARSA does not generate any target code. The application specification tool accepts a sequential program written in the SISAL functional language. The tool converts this code into a DAG, which is represented in textual form by the IF1 (Intermediate Form 1) acyclic graphical language. The architecture specification tool allows the user to interactively specify the target system in graphical form. The mapping and scheduling tool includes the HNF algorithm, the LC algorithm, and the LCTD algorithm. The performance assessment tool displays the expected runtime behavior of the scheduled program.

Casch can be considered a super set of various tools since it includes the major functionalities of these tools at a more advanced and comprehensive level, while offering additional useful features.

References

1. Cray Research Inc., UNICOS *Performance Utilities Reference Manual*, edition 6.0, 1991.
2. K. Kennedy, K.S. McKinley, and C. Tseng, "Interactive Parallel Programming Using the Parascope Editor," *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 3, Mar. 1991, pp. 329–341.
3. M. Wolfe, "The Tiny Loop Restructuring Research Tool," *Proc. Int'l Conf. Parallel Processing*, Vol. II, Aug. 1991, pp. 46–53.
4. C. Farhat and M. Lesoinne, "Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics," *Int'l J. Numerical Methods in Engineering*, Vol. 36, No. 5, 1993, pp. 745–764.
5. D. Pease et al., "PAWS (Parallel Assessment Window System): A performance Assessment Tool for Parallel Computing Systems," *Computer*, Vol. 24, No. 1, Jan. 1991, pp. 18–29.
6. M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 330–343.
7. T. Yang and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors," *Proc. 6th ACM Int'l Conf. Supercomputing*, ACM Press, New York,1992, pp. 428–433.
8. T.G. Lewis and H. El-Rewini, "Parallax: A Tool for Parallel Program Scheduling," *IEEE Parallel and Distributed Technology*, Vol.1, No. 3, May 1993, pp. 62–72.
9. V.M. Lo et al., "OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures," *Int'l J. Parallel Programming*, Vol. 20, No. 3, 1991, pp. 237–270.
10. B. Shirazi et al., "PARSA: A Parallel Program Scheduling and Assessment Environment," *Proc. Int'l Conf. Parallel Processing*, Vol. II, Aug. 1993, pp. 68–72.

many tasks that CASCH schedules to balance the load and minimize communication. In CASCH, a scheduling algorithm schedules the task graph generated based on this partitioning. Because one scheduling algorithm might not be suitable for a certain problem on a given architecture,[3] CASCH includes various algorithms that are suitable to various environments. Having a wide variety of algorithms in CASCH

- lets the user select a type of algorithm that is suitable to a particular architectural configuration;
- allows simultaneous comparisons among various algorithms, based on performance objectives such as schedule length, number of processors used, algorithm's running time, and so forth;
- lets the user compare the algorithms using manually generated task graphs and real data measured at execution time of a number of applications; and
- lets the user optimize the code for a given application program by running various scheduling algorithms to choose the best schedule.

Communication inserter

Communication primitives carry out synchronization among the tasks running on multiple processors. 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. CASCH can automatically insert these primitives, reducing the programmer's burden and eliminating insertion errors. The procedure for inserting communication primitive is as follows.

After scheduling and mapping, CASCH allocates each task to a processor. If an edge emerges from one task to another task that belongs to a different processor, CASCH inserts the *send* primitive after the task. Similarly, if an edge comes from another task in a different processor, CASCH inserts the *receive* primitive before the task.

The insertion method does not ensure a correct communication sequence because a deadlock might occur. Thus, we use a *send-first* strategy for a reordering of communication primitives. That is, we reorder *receives* according to the order of *sends*. Next we describe the communication primitive insertion algorithm

Assume that after scheduling and mapping, each task T_i of the task graph is allocated by the scheduler to processor $M(T_i)$, where M is a function mapping a task number to a processor number. For each edge e_k from task T_i to T_j for which $M(T_j) \neq M(T_j)$, insert a *send* primitive after task T_i in processor $M(T_i)$, denoted by $S(e_k, T_i, M(T_j))$; insert a *receive* primitive before task T_j in processor $M(T_j)$, denoted by $R(e_k, T_j, M(T_i))$. Once a message is assigned by the scheduler for transfer to a processor, eliminate other *sends* and *receives* that transfer the same message to the same processor. Now, for each processor, we have a sequence, $X(e_{m1}, T_{m1}, P_{m1})$, $X(e_{m2}, T_{m2}, P_{m2})$,…, where X could be either S or R.

```
/* For P0 */
     /* 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(P1, 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(P1, 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(P2, data[1][1]);
            InterMult(data[2][0],data[2][1],data[1][0],2);
            send(P2, 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 P1 */
            receive(P0, data[2][2]);
            receive(P0, 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 P2 */
            receive(P0, data[1][1]);
            IntraMult(data[1][1],data[0][1],2);
            receive(P0, 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 3. The parallel code for a Fast Fourier Transform.

For each pair of processors, say P_1 and P_2, extract all $S(e_{mi},T_{mi},P_2)$ from processor P_1 to form a subsequence S_{P1}, and extract all $R(e_{mj},T_{mj},P_1)$ from processor P_2 to form a subsequence R_{P2}. Then, within each segment of the subsequence S_P with the same task number, exchange the order of *sends* according to the order of *receives* as defined by the subsequence R_{P2}. If the two resultant subsequences still don't match, reorder R_{P2} according to the order of S_{P1}.

Code generation

We use the example in Figure 2 to illustrate our code-generation method. Figure 3 shows the generated parallel code for three processors (assuming $N = 8$). Note that we show only the main program for each processor. The data structure in Figure 3 is the same as in Figure 2. In Figure 3, processor P0 stores the initial data and it transmits data to other processors such that each processor obtains the portion of data required for its computation. Consequently, the memory space is compacted. To reduce the number of message transfers and, consequently, the time to initiate messages, CASCH will pack and send several messages together. For example, it can pack the first four messages into one message and send them to processor P0. CASCH also implements such optimizations. Finally, processor P0 receives the fourth data partition of the result, processor P1 receives the third data partition, and processor P2 receives the second data partition.

Graphical user interface

CASCH's graphical capabilities offer an easy-to-use window-based interactive interface. It presents 12 buttons, each mapping to a specific facility. We now discuss the 12 facilities.

- *Source*: users can create, edit, or browse through sequential programs. They can also generate a task graph from the user program.
- *DAGs*: displays a task graph (that is, a DAG) generated from the user program (Figures 4 and 7 show the DAGs for the FFT and Gaussian elimination programs respectively). Other options include displaying a randomly generated DAG or interactively creating a DAG. Zooming facilities (horizontally, vertically, or both) are included for proper viewing.
- *TIGs*: displays TIGs (task interacting graphs with undirected edges), similar in functionality to DAGs.
- *Processor network:* users can display a processor architecture (including the processors and the network topology). The editing facilities, similar to DAGs, let the user interactively create various network topologies (see Figure 5).
- *Scheduling*: includes a submenu from which the user must first select one of the three classes of the scheduling algorithms—BNP (bounded number of processors), UNC (unbounded number of clusters), and APN (arbitrary processor network). Within each class, the user needs to select one of the scheduling algorithms. The

scheduling algorithm requires the user to enter a number of parameters.

- *Show schedule*: displays the schedule generated after invoking a scheduling algorithm (see Figure 6). Clicking on any task in the Gantt chart displays its start and finish times. Also, the display shows the total schedule length in the right corner, and a schedule also includes communication messages on the network (displayed through another window, which you invoke by clicking on any two processors). An important feature of this facility is the trace option, which shows a step-by-step scheduling of each task. This is very useful for understanding the operation of a scheduling algorithm through observation of the order in which the algorithm schedules tasks. The program permits multiple charts to be opened concurrently, allowing for a comparison of the schedules various algorithms generated. In most cases, it is necessary to try different algorithms. Figure 7 shows a task graph generated by the program.
- *Mapping*: includes a number of mapping algorithms that can map TIGs onto the processors. CASCH includes algorithms based on A*, recursive clustering, and simulated annealing.[10] Some scheduling algorithms (such as UNC algorithms) might first generate clusters that need to be mapped onto the processors using one of these mapping algorithms.
- *Show mapping*: shows an assignment of tasks to the processors that a mapping algorithm generated. The display includes a TIG in which a processor number is attached to each task (indicating the processor number to which this task is allocated).
- *Code generation*: creates the parallel code for a given program according to a schedule or map generated by a scheduling and mapping algorithm.
- *Performance*: includes processor utilization, time spent in computation and communication, and speedup. The computation and communication timing results are obtained by inserting the *dclock()* procedure call before and after each intertask communication.
- *Data partitioning*: includes tools for displaying structured and unstructured meshes as well as partitioning of data across different processors.
- *Program testing and performance evaluation*: After CASCH generates the

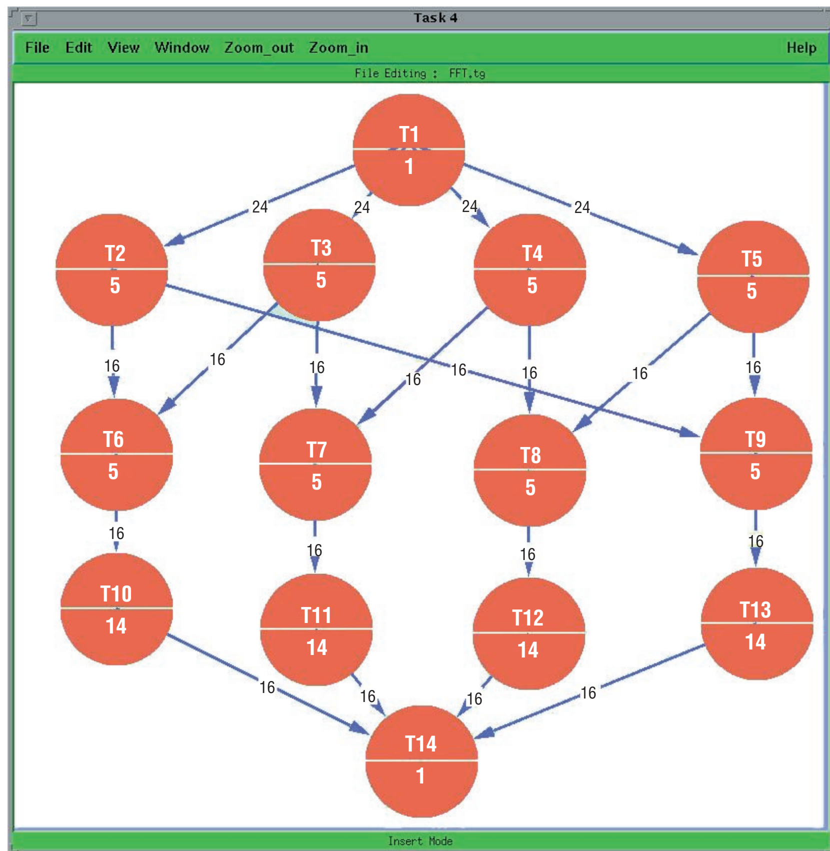


Figure 4. The directed acyclic graph for the FFT program.

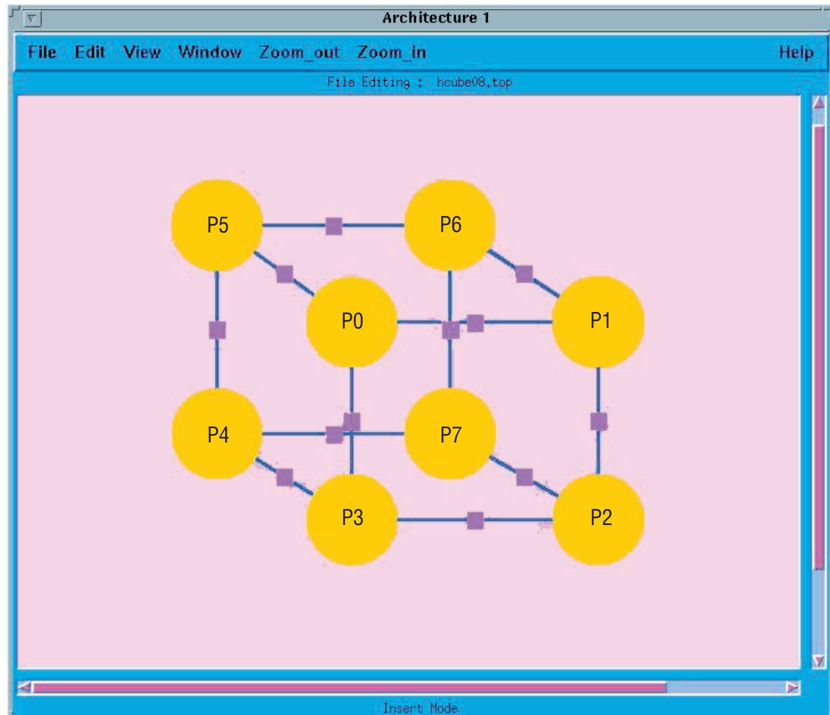


Figure 5. A processor graph.

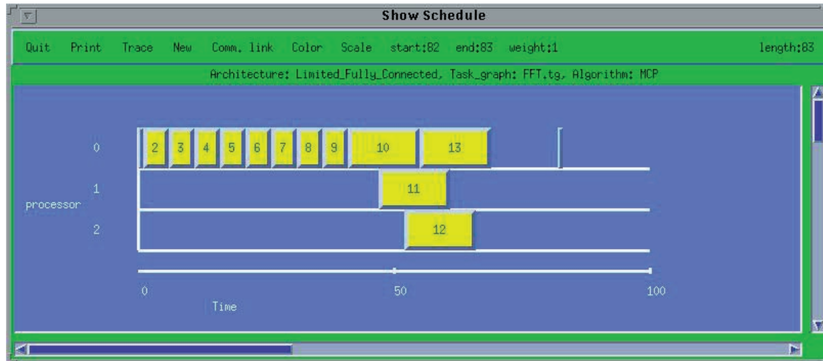


Figure 6. Display of the Gantt chart showing the schedules generated by the MCP and ETF algorithms for the FFT program (schedule length = 83).

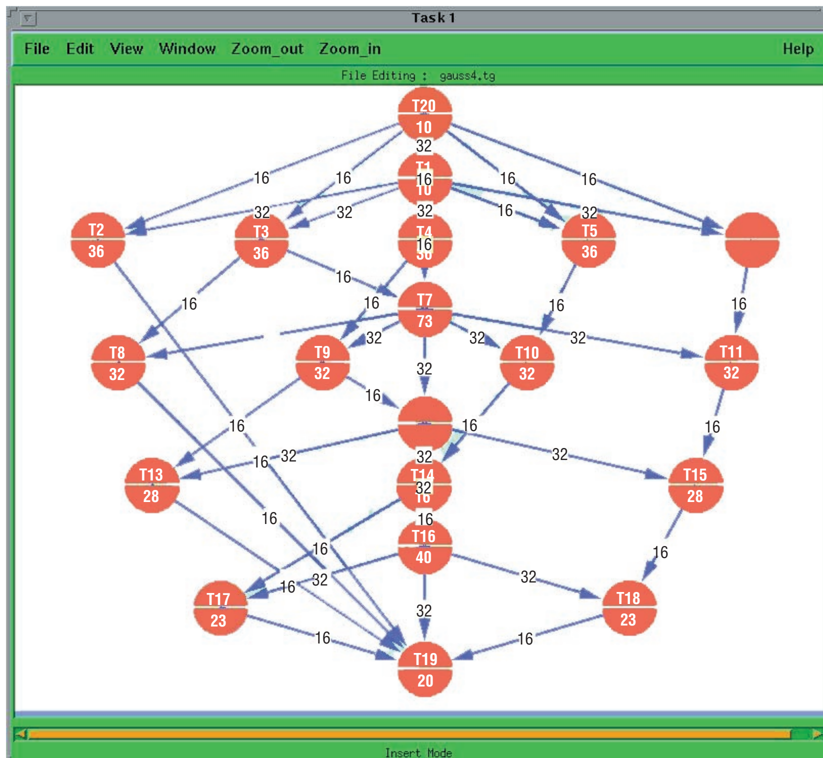


Figure 7. A Gaussian elimination task graph.

parallel code, the user can compile the resulting program to generate native parallel machine code on the target machine (for example, the IBM SP2) for testing. Because CASCH automatically inserts some statements for collecting runtime statistics, the user can use CASCH to parse these statistics to generate runtime profiles to visualize the behaviors of different parts of the program. By repeating the whole design process again, the user can improve his or her program.

Results

CASCH runs on a Sun workstation that is linked through a network to an Intel Paragon and an IBM SP2. We have parallelized several applications on CASCH by using some of the scheduling algorithms described earlier. Here we discuss some preliminary results obtained by measuring the performance of three applications: FFT, Laplace equation solver, and *N*-Body problem. These results demonstrate the viability and usefulness of CASCH as well as compare various scheduling algorithms. For reference, we include the results obtained with code generated by random scheduling of tasks. CASCH generates the target code by first partitioning the data among processors in a fashion that reduces the dependencies among the partitions. Based on this partitioning, an SPMD-based code is generated by randomly allocating the tasks to the processors. Hereafter, *randsch* denotes the results of these randomly scheduled programs.

The first set of results (see Table 2) is for the FFT example shown earlier with four different sizes of input data: 512, 1,024, 2,048, and 4,096 points. Table 2 shows the execution times for various data sizes using different scheduling algorithms. Each value is an average of 10 runs on the Intel Paragon and IBM SP2. The Paragon consists of 140 i860/XP processors with a clock speed of 50 MHz, while the SP2 consists of 48 160-MHz IBM P2SC processors.

We observe that the execution times vary considerably with different algorithms. Among the UNC algorithms, the DCP (Dynamic Critical Path) algorithm yields the best performance due to its superior scheduling method; it also yields the best performance overall. Among the BNP algorithms, MCP (Modified Critical Path) and DLS (Dynamic Level Scheduling) are in general better, primarily because of their better task priority assignment methods. Among the APN algorithms, BSA (Bubble Scheduling and Allocation) and MH (Mapping Heuristic) perform better, due to their proper allocation of tasks and messages. All algorithms perform better than randsch: Compared to the random scheduling, the level of performance improvement is up to 400%.

Our second application is based on a Gauss-Seidel algorithm to solve Laplace equations. The four matrix sizes used are

Table 2. Execution times of the FFT application for all the scheduling algorithms on the Intel Paragon and IBM SP2.

Algorithm		512 points Paragon	512 points SP2	1,024 points Paragon	1,024 points SP2	2,048 points Paragon	2,048 points SP2	4,096 points Paragon	4,096 points SP2
	randsch	2.65	2.06	7.14	5.50	22.11	17.05	62.58	48.50
UNC	DCP	0.64	0.50	0.85	0.66	1.27	0.98	1.74	1.34
	DSC	0.71	0.55	1.08	0.84	1.61	1.25	2.38	1.85
	EZ	0.78	0.60	1.47	1.14	2.25	1.74	3.81	2.95
	LC	0.81	0.62	1.46	1.12	2.34	1.80	3.82	2.92
	MD	0.73	0.56	1.23	0.95	2.20	1.70	3.68	2.84
BNP	ETF	0.76	0.59	1.23	0.94	2.09	1.60	3.44	2.62
	HLFET	1.40	1.09	3.00	2.29	6.77	5.17	13.38	10.19
	ISH	0.73	0.56	1.01	0.78	1.46	1.13	2.30	1.78
	LAST	0.86	0.66	1.95	1.48	4.29	3.33	8.49	6.46
	MCP	0.72	0.55	1.10	0.85	1.88	1.45	2.69	2.08
	DLS	0.73	0.56	1.27	0.99	1.87	1.45	3.21	2.46
APN	BSA	0.74	0.57	0.99	0.75	1.47	1.12	2.01	1.53
	BU	0.86	0.66	1.71	1.30	3.36	2.60	6.19	4.80
	DLS	0.79	0.61	1.21	0.94	1.94	1.49	3.26	2.53
	MH	0.72	0.56	1.13	0.88	1.83	1.40	2.60	1.98

Table 3. Execution times of the Laplace equation solver application for all the scheduling algorithms on the Intel Paragon and IBM SP2.

		Matrix dimension							
		8		16		32		64	
Algorithm		Paragon	SP2	Paragon	SP2	Paragon	SP2	Paragon	SP2
	randsch	2.89	2.24	24.12	18.80	72.32	56.08	259.00	199.65
UNC	DCP	1.06	0.81	6.08	4.68	16.02	12.45	75.01	57.59
	DSC	1.34	1.02	6.30	4.79	16.42	12.77	88.12	67.80
	EZ	1.44	1.12	6.95	5.31	18.28	14.17	94.03	73.34
	LC	1.25	0.97	6.95	5.38	18.81	14.57	100.52	76.89
	MD	1.25	0.97	6.52	5.01	17.08	13.31	83.09	63.98
BNP	ETF	1.25	0.97	6.73	5.14	17.75	13.56	80.31	61.64
	HLFET	1.34	1.03	7.60	5.88	32.71	25.01	195.99	149.61
	ISH	1.34	1.03	7.39	5.65	17.08	13.02	70.85	54.06
	LAST	1.54	1.18	7.17	5.58	19.87	15.46	106.75	81.51
	MCP	1.54	1.18	6.52	5.08	16.95	13.10	80.74	61.42
	DLS	1.34	1.03	6.30	4.79	17.22	13.28	94.25	72.53
APN	BSA	1.25	0.97	6.95	5.29	18.81	14.57	83.17	64.81
	BU	3.26	2.51	7.60	5.82	20.13	15.36	113.95	88.22
	DLS	1.92	1.47	8.69	6.78	23.25	17.99	108.75	83.36
	MH	1.92	1.48	8.69	6.73	23.25	17.70	110.34	84.55

8, 16, 32, and 64. The application execution times using various algorithms and data size are shown in Table 3. Again, using the DCP algorithm, more than 400% improvement over randsch is obtained. The UNC algorithms in general yield better schedules.

The third application is the N-Body problem. The execution times results are shown in Table 4. Again, the scheduling algorithms demonstrate a similar trend in application execution times on both parallel machines as in the previous two applications. The running times of the scheduling algorithms for the three applications are shown in Table 5. Here, we can see that some scheduling algorithms take much longer times than the others due to their higher complexities. (Details about the complexities of the algorithms are discussed elsewhere.[3]) For example, DCP, EZ (Edge Zeroing), MD (Mobility Directed), BSA, and DLS take two to five orders of magnitude longer to finish the scheduling. Thus, there is a trade-off between the performance and the speed of a scheduling algorithm. For example, the DCP algorithm can generate better solutions than the DSC algorithm, but it is slower.

It's important to note that the performance of the scheduling algorithms can have substantial variations. For instance, even though the average performance of the DCP algorithm is better overall, it does perform worse in some cases. Thus, the user might have to try different schedulers to obtain the best results for the application at hand.

Table 4. Execution times of the N-Body application for all the scheduling algorithms on the Intel Paragon and IBM SP2.

Algorithm		512 points		1,024 points		2,048 points		4,096 points	
		Paragon	SP2	Paragon	SP2	Paragon	SP2	Paragon	SP2
	randsch	11.45	8.87	26.20	20.11	78.18	60.74	229.04	177.09
UNC	DCP	3.13	2.43	4.29	3.32	6.01	4.68	9.86	7.60
	DSC	3.22	2.49	5.40	4.12	7.62	5.93	12.38	9.52
	EZ	3.80	2.90	7.26	5.55	10.44	8.08	17.34	13.19
	LC	4.52	3.52	7.10	5.44	12.16	9.38	18.60	14.19
	MD	3.90	3.02	5.90	4.54	10.65	8.30	20.65	16.02
BNP	ETF	3.91	3.05	6.78	5.15	11.54	8.94	15.99	12.35
	HLFET	7.23	5.58	15.37	11.75	39.92	30.46	68.04	52.12
	ISH	3.48	2.69	5.34	4.06	6.89	5.37	12.48	9.60
	LAST	4.88	3.77	11.50	8.94	26.11	20.36	50.22	38.47
	MCP	3.50	2.66	5.24	4.08	9.57	7.42	14.01	10.71
	DLS	3.79	2.89	6.64	5.11	9.84	7.53	16.44	12.51
APN	BSA	3.18	2.47	4.59	3.57	7.72	5.89	10.00	7.71
	BU	4.55	3.48	8.29	6.32	17.90	13.73	30.24	23.11
	DLS	3.64	2.78	6.14	4.68	9.48	7.36	15.82	12.25
	MH	3.75	2.87	5.23	4.08	9.08	6.90	11.79	9.18

WE ARE CURRENTLY working on extending CASCH's capabilities by

- supporting 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, global variable checking, and so forth;
- designing and implementing partitioners for automatic or interactive program partitioning;
- designing an intelligent tool that will select an appropriate scheduling algorithm for a given application; and
- enhancing the task graph module so that huge task graphs (for example, for Laplace equation of a larger matrix size) can be handled; in this regard, we are considering the implementation of the parameterized task graph technique proposed by Cosnard and Loi.[11]

Acknowledgments
We thank the referees for their constructive and insightful comments that have greatly improved the presentation of this article. We presented preliminary versions of portions of this article at the 1997 International Conference on Parallel Processing and at the Third European Conference on Parallel Processing. The Hong Kong Research Grants Council supported this research under contract numbers HKUST RI 93/94.EG06 , HKUST734/97E, and HKU7124/99E.

References

1. E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.
2. Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 7, No. 5, May 1996, pp. 506–521.
3. Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel and Distributed Computing*, Vol. 59, No. 3, Dec. 1999, pp. 381–422.
4. S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing*, Vol. II, Aug. 1988, pp. 1–8.
5. E. Lord, J.S. Kowalik, and S.P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," *J. ACM*, Vol. 30, No. 1, Jan. 1983, pp. 103–117.
6. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.
7. Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2," *IEEE Parallel and Distributed Technology*, Vol. 4, No. 1, Spring 1996, pp. 9–23
8. W.W. Chu, M.-T. Lan, and J. Hellerstein, "Estimation of Intermodule Communication (IMC) and Its Applications in Distributed Processing Systems," *IEEE Trans. Computers*, Vol. C-33, No. 8, Aug. 1984, pp. 691–699.
9. A. Ghafoor and J. Yang, "A Distributed Heterogeneous Supercomputing Management System,"*Computer*, Vol. 26, No. 6, June 1993, pp. 78–86.
10. T.M. Nabhan and A.Y. Zomaya, "A Parallel Simulated Annealing Algorithm with Low Communication Overhead," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 12, Dec. 1995, pp. 1226–1233.
11. M. Cosnard and M. Loi, "Automatic Task Graphs Generation Techniques," *Parallel Processing Letters*, Vol. 5, No. 4, Dec. 1995, pp. 527–538.

Ishfaq Ahmad is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology. His research interests are in the areas of parallel programming tools, scheduling and mapping algorithms for scalable architectures, video compression, and interactive multimedia systems. He is director of the Multimedia Technology Research Center at HKUST, where he and his colleagues are working on a number of research projects related to information technology, in particular in the areas of video coding and interactive multimedia. He received a BSc in electrical engineering from the University of Engineering and

Table 5. Scheduling times (seconds) for the applications on a SPARC Station 2 for all the scheduling algorithms.

(a) FFT application.

Algorithm		Number of points			
		512	1,024	2,048	4,096
UNC	DCP	15.80	69.54	304.50	1,306.32
	DSC	0.09	0.22	0.53	1.28
	EZ	15.37	71.18	336.44	1540.80
	LC	0.09	0.23	0.60	1.54
	MD	69.78	543.16	4,147.10	32,764.68
BNP	ETF	0.16	0.56	1.99	6.90
	HLFET	0.15	0.44	1.27	3.78
	ISH	0.13	0.45	1.57	5.47
	LAST	0.30	1.21	4.83	19.17
	MCP	0.14	0.43	1.35	4.24
	DLS	0.29	1.03	3.64	12.79
APN	BSA	5.38	29.61	159.71	855.99
	BU	0.41	1.14	3.06	8.30
	DLS	93.35	512.14	2,809.33	15,521.60
	MH	4.58	20.62	93.28	417.03

(b) Laplace equation solver application.

Algorithm		Matrix dimension			
		8	16	32	64
UNC	DCP	7.03	9.60	44.95	198.15
	DSC	0.04	0.10	0.29	0.69
	EZ	7.15	9.71	35.00	164.50
	LC	0.07	0.09	0.16	0.41
	MD	7.65	10.01	111.99	864.95
BNP	ETF	0.07	0.10	0.30	1.04
	HLFET	0.09	0.11	0.29	0.86
	ISH	0.08	0.09	0.35	1.19
	LAST	0.10	0.15	0.82	3.24
	MCP	0.04	0.10	0.19	0.59
	DLS	0.07	0.10	0.36	1.29
APN	BSA	2.81	4.92	27.99	148.62
	BU	0.37	0.50	0.73	1.98
	DLS	7.81	10.86	75.90	422.22
	MH	2.41	3.25	16.29	74.75

(c) N-Body application.

Algorithm		Number of points			
		512	1,024	2,048	4,096
UNC	DCP	6.65	14.92	289.61	1,472.51
	DSC	0.05	0.11	0.23	0.58
	EZ	7.77	17.38	350.19	1,856.19
	LC	0.06	0.08	0.32	0.79
	MD	7.57	42.58	3,133.97	19,292.11
BNP	ETF	0.08	0.22	2.28	8.03
	HLFET	0.07	0.16	0.75	2.20
	ISH	0.07	0.09	0.32	1.19
	LAST	0.09	0.26	2.65	9.06
	MCP	0.07	0.10	0.45	1.22
	DLS	0.08	0.31	3.09	11.90
APN	BSA	2.05	14.91	248.36	1,279.62
	BU	0.40	0.47	1.11	3.06
	DLS	17.79	312.87	8,105.45	36,430.02
	MH	1.52	11.48	368.57	1,788.90

Technology, Lahore, Pakistan and an MS in computer engineering and a PhD in computer science, both from Syracuse University. He is a member of the IEEE Computer Society. Contact him at the Dept. of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong; iahmad@cs.ust.hk; www.cs.ust.hk/faculty/iahmad.

Yu-Kwong Kwok is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. His research interests include software support for parallel and distributed computing, mobile computing, heterogeneous cluster computing, and distributed multimedia systems. He received a BSc in computer engineering from the University of Hong Kong and an MPhil and a PhD in computer science from the Hong Kong University of Science and Technology. He is a member of the IEEE Computer Society and the ACM. Contact him at the Dept. of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong; ykwok@eee.hku.hk; www.eee.hku.hk/~ykwok.

Min-You Wu is an associate professor in the Department of Electrical and Computer Engineering at the University of Central Florida. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He received an MS from the Graduate School of Academia Sinica, Beijing, China and a PhD from Santa Clara University, California. He is a senior member of the IEEE and a member of the ACM. Contact him at the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816-2450, USA; wu@ece.engr.ucf.edu; www-ece.engr.ucf.edu/~wu.

Wei Shu is an associate professor in the Department of Electrical and Computer Engineering at the University of Central Florida. Her current interests include dynamic scheduling, resource management, runtime support systems for parallel and distributed processing, multimedia networking, and operating system support for large-scale distributed simulation. She received a BS from Hefei Polytechnic University, China, an MS from Santa Clara University, and a PhD from the University of Illinois at Urbana-Champaign. She is a member of the ACM and the IEEE. Contact her at the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816-2450, USA; shu@ece.engr.ucf.edu; www-ece.engr.ucf.edu/~shu.