Cluster Comput (2006) 9:355–371
DOI 10.1007/s10586-006-9747-2

A taxonomy of application scheduling tools for high performance cluster computing*

Jiannong Cao · Alvin T. S. Chan · Yudong Sun · Sajal K. Das · Minyi Guo

Received: 1 September 2003 / Revised: 1 March 2004 / Accepted: 1 May 2004

Abstract Application scheduling plays an important role in high-performance cluster computing. Application scheduling can be classified as job scheduling and task scheduling. This paper presents a survey on the software tools for the graph-based scheduling on cluster systems with the focus on task scheduling. The tasks of a parallel or distributed application can be properly scheduled onto multi-processors in order to optimize the performance of the program (e.g., execution time or resource utilization). In general, scheduling algorithms are designed based on the notion of task graph that represents the relationship of parallel tasks. The scheduling algorithms map the nodes of a graph to the processors in order to minimize overall execution time. Although many scheduling algorithms have been proposed in the literature, surprisingly not many practical tools can be found in practical use. After discussing the fundamental scheduling techniques, we propose a framework and taxonomy for the scheduling tools on clusters. Using this framework, the features of existing scheduling tools are analyzed and compared. We also discuss the important issues in improving the usability of the scheduling tools.

Keywords Scheduling tool · Cluster · Task scheduling · Task graph · Directed acyclic graph

1 Introduction

Many large-scale applications in science, engineering and commerce should be executed on parallel systems to achieve high performance computing. The rapid advances in powerful microprocessors, high-speed networks and standard software tools are enabling the clusters of computers to be a cost-effective substitute for parallel computers. Thus, parallel computing has in large extent been migrating from expensive high-end supercomputers to lower-cost clusters built with commodity-off-the-shelf (COTS) computers and commonly used software [6, 10].

A cluster is composed of multiple standalone computers connected via a network. To exploit the system capability and implement high-performance computing on it, software supports are required to realize the consolidated computational capability. The software supports can be implemented at various levels such as operating systems (e.g., Solaris MC [9], GLUnix [23], and MOSIX [40]), resource management systems (e.g., Condor [13], LSF [39], MARS [21], and Legion [38]), and parallel programming environments (e.g., PVM [45], MPI [41], and OpenMP [43]). Importantly, the soft-

*This work is supported by the Hong Kong Polytechnic University under grant H-ZJ80 and by NASA Ames Research Center by a cooperative grant agreement with the University of Texas at Arlington.

J. Cao (✉) · A. T. S. Chan
Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong
e-mail: csjcao@comp.polyu.edu.hk

A. T. S. Chan
e-mail: cstschan@comp.polyu.edu.hk

Y. Sun
School of Computing Science, University of Newcastle upon Type, Newcastle upon Type, NE1 7RU, UK
e-mail: yudong.sun@ncl.ac.uk

S. K. Das
Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019-0015, USA
e-mail: das@cse.uta.edu

M. Guo
Department of Computer Software, University of Aizu, Aizu-Wakamatsu City, Fukushima 965-8580, Japan
e-mail: minyi@u-aizu.ac.jp

ware support for application scheduling is critical to realize high-performance parallel computing. The application scheduling aims to appropriately allocate parallel programs to processors so that the resource utilization can be improved and the execution time can be reduced. Application scheduling can be performed at job level or task level. *Job scheduling* [4, 18, 34] deals with the allocation of *independent* programs or tasks to processors according to the priorities of the jobs and the availability of the resources. At present, job scheduling is widely used for scheduling parallel programs on clusters. A parallel job is submitted with the required number of processors. Job scheduler assigns the job to a suitable queue. When the required processors become available, the job is dispatched to run on them. Job scheduling aims at balancing the workload among the processors and therefore optimizing the system-wide throughput. It maps the processes of a parallel job to the processors without the consideration of the dependency between the tasks. A large number of commercial job scheduling systems are available, including Condor [13], LSF [39] and Loadleveler [30]. The surveys of job scheduling techniques can be found in [4, 5, 18–20, 34].

Differently, *task scheduling* handles the allocation of *dependent* tasks of a parallel program to the processors in order to minimize the overall execution time [22, 35, 36, 47, 49, 59]. As an attempt to design a task scheduling method for a graph-oriented programming environment on clusters [12], we face the problem of formulating an effective scheduling approach and implementing the support tool. This requirement motivates us to survey the literature about the scheduling tools for cluster computing. Since task scheduling is usually based on the relationship between the tasks, which can be modeled as a *task graph*, our survey emphasizes on the graph-based task scheduling tools.

Although enormous task scheduling algorithms have been proposed [15, 35, 36, 47], very few task scheduling tools can be found on cluster systems. Moreover, most of the researches in this area are theoretical work rather than practical implementation. Only a few tools are implemented for practical use. In this paper, we propose a framework and taxonomical classification of scheduling tools to assist the design of the tools. After the discussion on the requirements and fundamental techniques of task scheduling, the features of representative scheduling tools are summarized and compared using the proposed taxonomy. We also discuss the important aspects in improving the usability of the scheduling tools. Our framework can provide a guideline for developing scheduling tools on clusters.

The rest of the paper is organized as follows. Section 2 is an overview of the architecture, applications and task scheduling tools of cluster computing. Section 3 introduces the general task scheduling techniques. Section 4 describes our framework and taxonomy of the scheduling tools. Section 5 discusses the representative scheduling tools and compares their characteristics. Section 6 concludes the paper and discusses the future research issues.

2 Task scheduling on clusters

Cluster systems have been widely used to process parallel applications in various fields such as scientific computing, image processing, artificial intelligence, physical and biological modeling, databases, and Web servers. With the rapid development in hardware and software, a cluster system is now capable of performing large-scale computations that were conventionally feasible only on supercomputers. Task scheduling plays an important role in enhancing this capability of clusters.

2.1 Architectural features

In a cluster system, standalone computers are consolidated into a single, unified system to perform parallel computing [10, 29, 44]. Different from traditional parallel computers, each machine in a cluster can be independently in operation and be separately accessed by different users and applications. Also different from distributed systems, the machines in a cluster are integrated into a unified single resource. A user can access the cluster through a single interface. Applications can be launched on any machine in the cluster and dispatched to run on more processors as in a parallel computer.

Figure 1 shows the generic architecture of a cluster system in which the computer nodes can be PCs, workstations, SMPs (Symmetric Multiprocessors) and even supercomputers. The nodes are interconnected via a commodity high-speed network such as Fast Ethernet or Myrinet [42]. The operating system (OS) on each node can be multi-user, multitasking, and multi-threaded systems such as Linux, Solaris, and Windows NT. A cluster can be a *homogeneous system* in which all nodes have similar architecture and run the same

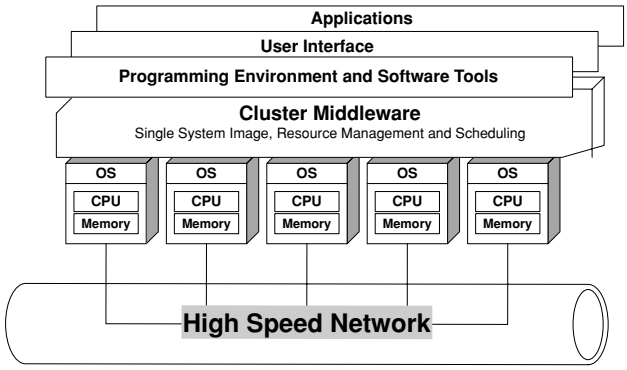


Fig. 1 Cluster architecture

OS. A cluster can also be a *heterogeneous system* where the nodes have different architecture and run different OS. A cluster middleware layer is constructed on top of the operating systems to create a single system image (SSI) [10] over the cluster and to manipulate resource management and scheduling. Programming environments, software tools, and user interfaces are settled upon the middleware layer to support application development. Communications between the nodes are implemented based on message passing. In a cluster, the inter-process communication latency is higher than that in a shared-memory parallel computer or in the multiprocessors linked by proprietary interconnection network. The high communication latency will influence the performance of cluster computing.

Therefore, task scheduling is an important technique to exploit the consolidated computing power, reduce the communication overhead, and minimize the execution time. The task scheduling is based on the computation and communication costs of the tasks, the speed and workload of the processors, and the bandwidth of network. The scheduling tools usually consist of the components existing in different layers from user interface to cluster middleware. A scheduling tool requires a user interface to input user program and specify the scheduling requirements. Program preprocessors are needed to process the program code or the task graph to determine a scheduling scheme. The scheduling is eventually performed by the cluster middleware. The scheduling tools may also support the execution profiling, performance analysis and visualization.

2.2 Application requirements

Compute-intensive applications highly rely on the task scheduling to realize high-performance computing on clusters. The requirements of task scheduling can be identified in the major application fields of cluster computing.

1. Scientific Computing

Scientific computing covers a wide range of applications, including matrix computation, linear system solver, Fast Fourier Transform (FFT), Partial Differential Equation (PDE) solver and other applications [11]. For example, partial differential equations can be used to describe the behavior of a physical system such as in computational fluid dynamics (CFD) [33]. The PDE algorithms are massively parallel computing problems with the computational complexity at least an order of magnitude beyond the capabilities of today's workstations in memory requirement and CPU time. However, a cluster of workstations can provide adequate capacity to run the PDE solvers. The task scheduling can produce a proper mapping of the PDE algorithms to the processors so that high performance computing can be achieved.

Scientific computing applications may also include heavy inter-processor communication that restricts the performance of these applications. Thus, task scheduling is required to determine the task decomposition and mapping that can reduce the communication overhead.

2. Image Processing

Image processing is an application area with potentially high parallelism [3, 51]. For example, ray tracing is a graphical rendering algorithm that creates an image from the description of the objects [50]. Parallel ray tracing usually uses two methods to parallelize the rendering operations: *image parallel* and *object-space parallel* [11]. In the image parallel method, a certain number of rays are assigned to each processor. Each processor possesses the description of all objects but renders a section of the image. In the object-space parallel method, the description of all objects is partitioned into sub-volumes that are distributed to the local memory of each processor. Each processor computes all the ray tracing related to that sub-volume. Due to the imbalanced workload in parallel rendering, dynamic scheduling is required to balance the workload on the processors.

3. System Modeling and Simulation

System modeling and simulation studies the features and evolution of physical, biological, electrical, mechanical and social systems. In system modeling, dynamic scheduling is required to redistribute the imbalanced workload caused by the system evolution. For example, the *N*-body problem [53] is an application that simulates the evolution of the physical systems in astrophysics, plasma physics, molecular dynamics, fluid dynamics and other areas. A physical system contains numerous bodies that impose force influences on one another. The aggregated force influence results in the continuous evolution of the system. Running on a cluster, the *N*-body simulation incurs high communication for the data exchange between the processors. In addition, the computational workload is dynamically changing on the processors due to the system evolution. Thus, dynamic scheduling is required to balance the workload and reduce the communication overhead.

4. Optimization Problems

Optimization problems are a class of compute-intensive problems. These problems are solved by searching and evaluating a set of possible solutions to find the optimal solution that satisfies some problem-specific criteria [24]. Genetic algorithms (GA) [11], for example, are the optimization methods based on the evolutionary process of Darwinian natural selection and population genetics. A solution is represented by a set of parameters, usually a string of values called a

chromosome; each chromosome represents an individual. The search process is directed by a fitness function that is a measure of the quality of the evolution to find an optimal or good feasible solution. The workload of the search operations is not predetermined. Dynamic scheduling is required to balance the workload on parallel processors during the search.

5. Database Systems

Clusters are increasingly used to support database applications such as data mining and pattern matching [11, 54]. For a large database, the data can be distributed to the processing nodes and disks in a cluster. Database applications introduce the concepts of data placement and redistribution. Task scheduling can be used to distribute the queries to the nodes. The task scheduling should also handle the dynamic reorganization of a database called data replacement, which moves data between processing nodes to optimize the performance of query operations.

Many other applications can also be found in cluster computing. For example, cluster systems have been used as enterprise and Internet servers to provide high availability, reliability and scalability in business and e-commerce services [31, 52, 57].

2.3 Tool support for application scheduling

From the perspective of architecture and applications, the scheduling techniques on clusters should suit the features of cluster computing such as distributed, heterogeneous resources and high communication cost. Firstly, a cluster is a collection of computers with distributed architecture. The scheduling algorithms should adopt a distributed strategy to improve the efficiency and reliability of the scheduling procedure. On contrary, traditional parallel systems usually use the centralized scheduling where a single scheduler is responsible for the scheduling of all tasks to all processors.

Secondly, a cluster may consist of heterogeneous nodes with varied performance. The scheduling should be adaptive to the working nodes. A scheduling tool should operate with the support of the resource management subsystem (RMS) that monitors the available resources, starts the execution of applications and supports process migration [10]. A scheduling decision depends on the information provided by the RMS.

Furthermore, the communication over commodity networks has higher latency than the proprietary networks in supercomputers. The scheduling algorithms should also consider how to reduce the communication overhead between the tasks.

The supporting tools can also be designed by other means. For example, a performance analysis tool collects the execution profile to analyze the feature of a program. The execute profile can be used to guide the adaptive scheduling in the future execution of the same program.

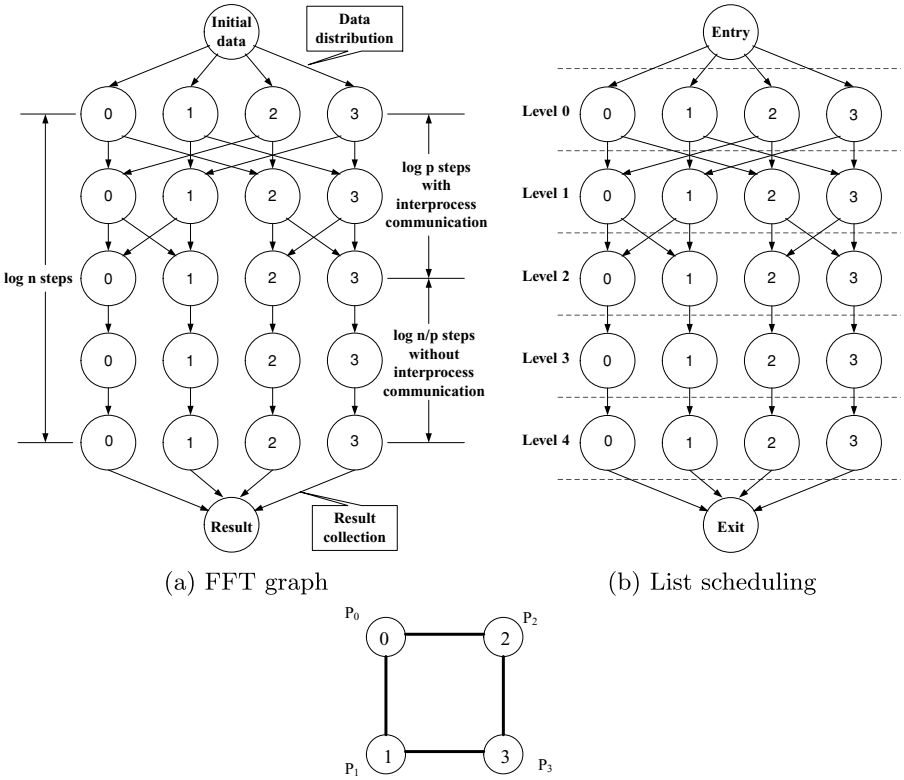
3 Task scheduling techniques

Various task scheduling algorithms have been proposed for parallel computing. Generally, there are two scheduling models: *static scheduling* and *dynamic scheduling* [32, 36, 47]. Static scheduling is performed at compile time provided that the characteristics of an application such as execution time, communication cost, data dependency, and synchronization requirement are known in advance [36, 47]. It allocates the tasks to individual processors before execution and the allocation remains unchanged during the execution. Dynamic scheduling conducts the scheduling at run time [14, 25] and the tasks can be reallocated during the execution. Dynamic scheduling can support dynamic load balancing and fault tolerance. It is certain that the dynamic scheduling operations introduce additional overhead to the program execution. So, dynamic scheduling algorithms should endeavor to reduce this overhead.

As a combination of static and dynamic scheduling, *hybrid scheduling* [47] includes two scheduling phases. First, static scheduling is made based on the estimated performance of a program. Then, dynamic scheduling is performed at run time to adjust the static task allocation to balance the workload on the processors.

3.1 Task graph

Task graph is a general model that describes the structure of a program for the purpose of scheduling. In a task graph, the nodes represent the computational tasks and the edges represent the relations between the tasks. A task scheduling algorithm maps the nodes to a set of processors in a form that can minimize the entire execution time of the program (called *schedule length*). As the optimal task scheduling is an NP-complete problem, many heuristics have been proposed to make the scheduling solvable in polynomial time complexity [15, 35, 36, 47]. These heuristics are the assumptions about the characteristics of parallel programs and parallel systems. Some heuristics assume that every task has the same computation cost but some heuristics allow arbitrary computation cost for each task. Some heuristics ignore the inter-task communication cost and some heuristics allow arbitrary communication cost. Some heuristics suppose an application to be run on an unlimited number



(a) FFT graph (b) List scheduling

(c) Node-to-processor mapping

Fig. 2 Directed acyclic graph and scheduling of a 16-point FFT on four processors

of processors but some are based on a limited number of processors.

Task scheduling algorithms are mainly designed based on *Directed Acyclic Graph* (DAG) [36, 47]. The DAG has deterministic structure on which deterministic scheduling algorithms can be designed. In a DAG, each node represents a task which in turn contains a set of operations that will be executed sequentially. When all input data to a node have arrived to it, the node can be triggered to execution. A node with no parent is called *entry node*. A node with no child is called *exit node*. The *weight* of a node is the *computation cost* of the task. The directed edges represent the precedence of the tasks. The edges determine a partial order of the execution flow. The *weight* of an edge is the *communication cost* between the adjoining nodes. The communication cost appears only when two adjoining nodes are executed on different processors. Otherwise, the communication cost will be zero if two nodes are allocated to the same processor. Please refer to [14, 35, 36, 47] for the details about the DAG-based scheduling algorithms.

Figure 2(a) shows a DAG of parallel Fast Fourier Transform (FFT) with 16 points running on four processors. As discussed in [24], the FFT performs a linear transformation that maps n sampled points, $X = \langle X[0], X[1], , X[n-1]\rangle$,from a cycle of a periodic signal onto an equal number of points, $Y = \langle Y[0], Y[1], , Y[n-1]\rangle$, that represent the frequency spectrum of the signal, where $Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki}$, $0 \leq i < n$ and $\omega = e^{2\pi\sqrt{-1}/n}$. The transform can be computed in $\log n$ iterations. Each iteration performs n complex multiplications and additions.

In parallel FFT algorithms such as the binary-exchange algorithm given in [24], the n points are evenly distributed to p processors by which every n/p contiguous points are assigned to one processor. In each of the iterations, each processor computes n/p complex multiplications and additions. Then, each pair of processors with binary labels only different in the mth most significant bit exchanges n/p complex values, where m is the iteration level from 0 to $\log n - 1$. In Fig. 2(a), the DAG of parallel FFT shows the iterative transform where $n = 16$ and $p = 4$. The size of the task graph is related to the number of processors p. The number of nodes at each level equals the number of processors to run the program. Each node represents the computational task of n/p complex multiplications and additions. The edges denote the data exchange between the iterations. The entry node distributes n points to p processors. The exit node collects the transform results from all processors.

3.2 Scheduling algorithms

The main algorithms for the DAG-based scheduling are *list scheduling* and *clustering*. Other algorithms also include *task duplication* that allows the processors to run duplicated instances of the tasks. These algorithms are discussed below.

3.2.1 Heuristics

As discussed in Section 3.1, the scheduling algorithms make various heuristics on a task graph and system architecture to simplify the algorithms in order that a scheduling can be determined in a reasonable time complexity [35, 36, 47, 55]. The heuristics can be made in the following aspects:

- **Task graph**: a task graph is allowed to possess an *arbitrary structure* or restricted to a *specific structure* (e.g., a tree).
- **Computation cost**: the nodes in a graph can have either *arbitrary* or *uniform* computation cost.
- **Communication cost**: the edges can have *arbitrary*, *uniform*, or *all-zero* communication cost. All-zero cost means that the communication is negligible.
- **Processors**: an *unlimited* or *limited* number of processors are usable to run a program.
- **Architecture**: the processors in a parallel system can be *fully connected* or in a *specific interconnection topology*. The architecture will influence the decision of task-to-processor mapping strategy.

3.2.2 List scheduling

List scheduling is a commonly used method for the DAG-based scheduling [36, 47]. It assigns priority to each node of a task graph and then allocates the nodes to the processors based on the priority. The nodes with high priority will be scheduled first.

The priority of a node can be determined in different ways. Also, different strategies can be used to map the nodes to the processors. The priority of a node is usually calculated based on two attributes: *t-level* (top level) and *b-level* (bottom level) [35, 36]. The *t-level* of a node is the length of the longest path from an entry node to the node (excluding the node itself). The length of a path is defined as the sum of the weights of the nodes and the weights of the edges in the path. The *t-level* is related to the earliest start time of a node. The *b-level* of a node is the length of the longest path from the node to an exit node. The *b-level* is bounded with the length of the *critical path* (CP), i.e., a path from an entry node to an exit node with the maximum length. There are different ways to determine the *b-level*. Most DAG-based scheduling algorithms examine a node for scheduling only when all the precedent nodes of the node have been scheduled. Nevertheless, some algorithms allow the scheduling of a child node before its parents. In such a case, the *b-level* becomes a dynamic attribute.

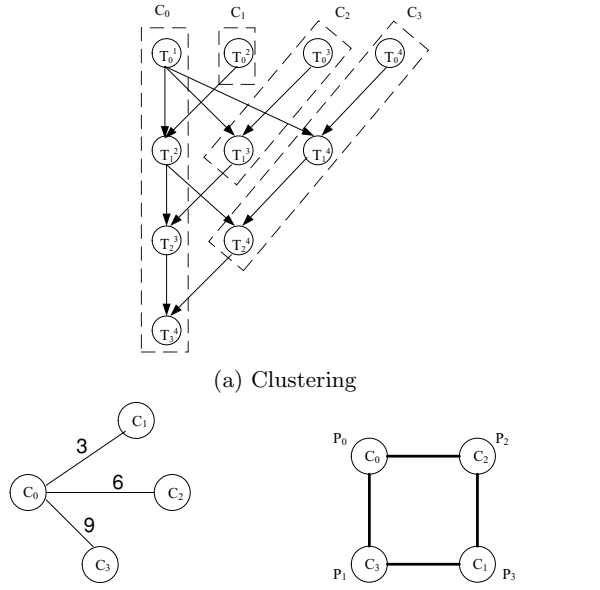
DAG-based scheduling algorithms use *t-level* or *b-level*, even both to decide the priority of a node. Some algorithms assign a higher priority to a node with a smaller *t-level*. Other algorithms assign a higher priority to a node with a larger *b-level*. Also, some algorithms assign a higher priority to a node with a larger difference between two levels, i.e., (*b-level* − *t-level*).

By list scheduling, the FFT problem in Fig. 2(a) can be scheduled to run on four processors as shown in Fig. 2(b). The nodes and edges can be viewed with uniform computation cost and communication cost. Thus, the nodes on the same level have the same *t-level* and *b-level*. The entry node can run on any of the processors to start the program. It distributes the points to the four nodes on level 0. The latter are scheduled to run on four processors. After the completion of the computational tasks, the nodes exchange the data. Next, the nodes on level 1 are scheduled to the processors. The nodes with the same label on all levels will be allocated to the same processor to make use of local data left by the precedent nodes. The procedure continues until the transform has finished. Finally, the exit node running on one processor collects the transform results. If the topology of four processors is a hypercube, the nodes can be mapped to the processors as shown in Fig. 2(c). The mapping guarantees that the nodes with data exchange are allocated to the neighboring processors so that the runtime communication latency can be minimized.

3.2.3 Clustering

Clustering is another approach for the DAG-based scheduling. Here, the *cluster* means a group of tasks. Clustering is the process that merges the nodes in a graph into clusters [47, 48, 59]. The tasks in the same cluster will be allocated together to a processor. Clustering is a two-phase scheduling procedure: *merging* the nodes into clusters and *mapping* the clusters to processors. The mapping phase is fulfilled by a sequence of optimization steps that include: (1) *cluster re-merging*: if the number of clusters is greater than the number of processors, the clusters will be further merged; (2) *task ordering*: if the tasks in a cluster are related by a precedence, the execution order of the tasks is arranged based on the precedence.

Usually, a clustering algorithm starts from an initial clustering and proceeds with a sequence of refinements on the clusters [47]. In the initial clustering, each task is viewed as a separate cluster. The clusters are refined by merging two adjacent clusters so as to remove the edge between them to reduce the communication weight. In the refinement phase, the



(a) Clustering

(b) Communication cost (c) Cluster-to-processor mapping

Fig. 3 Clustering on the DAG of Gauss-Jordan method

clusters are merged into larger clusters to reduce the overall execution time (i.e., schedule length).

For example, Fig. 3 shows a multistage clustering algorithm proposed in PYRROS [22, 59] for scheduling the Gauss-Jordan method, i.e. an elimination method for solving linear systems $Ax = b$. Figure 3(a) is the DAG of the Gauss-Jordan method. Node T_k^j (where $j \neq k+1$) represents a computational task with input data being columns k and j of the coefficient matrix and the output data being the modified column j. Node T_k^{k+1} is a broadcast node that sends column $k+1$ to all T_{k+1}^j nodes. To run the DAG on four processors, the nodes are merged into four clusters enclosed in the dotted boxes as shown in Fig. 3(a). Each cluster will be mapped to one processor. The clustering algorithm will be further discussed in Section 5.2.

3.2.4 Task duplication

Task duplication is a special scheduling method that duplicates selected tasks to run on more than one processor to reduce the inter-processor communication [1, 16, 46]. The duplication of the tasks aims to utilize the spare time slots on certain processors. The approach is conceived based on the fact that some child nodes must wait for the output from the parent nodes running on other processors. If the processors remain idle at different time slots, the spare time can be used to run the duplicated tasks of these parent nodes. Therefore, the parent and child nodes can be executed on the same processor and the output can be locally fed from parent to child without inter-processor communication. Some critical tasks can even have multiple instances running on more processors to further reduce the communication cost.

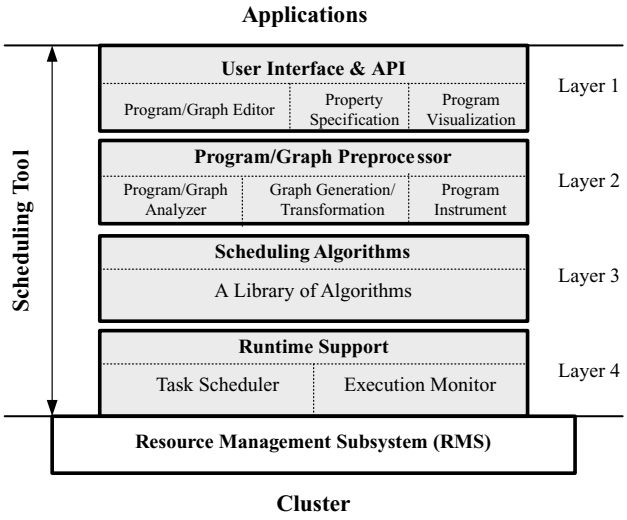


Fig. 4 The framework of scheduling tools

4 A framework and taxonomy of scheduling tools

Based on the discussion about cluster systems and task scheduling, a framework of scheduling tools can be built and a taxonomy of the tools can be specified.

4.1 Framework

Figure 4 shows the general framework of the scheduling tools. As discussed in Section 2.3, a scheduling tool is built on the resource management subsystem (RMS) that provides the information about the resources and their performance in a cluster. A scheduling tool is a middleware that links the applications to a cluster system. The framework consists of four layers. The components in each layer implement the specified functionalities in the scheduling.

Layer 1: User Interface & API

The user interface provides *program editor* and/or *graph editor* for user to input an application in textual or graphical form. The *property specification* component allows a user to describe the computation and communication costs of the application. The user can also define own scheduling algorithm for the application. The *program visualization* component is used for displaying the scheduling-related information such as a task graph, the architecture of a cluster, and the performance of an application. A user program can call the API provided by the scheduling tool to perform scheduling-related or other operations. For example, the ATME tool discussed in Section 5.5 provides execution profiling primitives

that can be instrumented into a user program to probe the runtime statistic data. The statistic data can be used to determine the scheduling of the program in later execution. The VDCE tool discussed in Section 5.1 provides task libraries that can be called in user applications to perform different operations such as computation, communication and control.

Layer 2: Program/Graph Preprocessor
When an application is provided in the form of text or graph, the *program/graph analyzer* will analyze the structure of the program and convert it into a task graph. The analyzer may be able to estimate the computation and communication costs by analyzing the operations in the program. For a textual input, the *graph generation* component creates a task graph of the program. For a graphical input, the *graph transformation* converts the original graph into a task graph that is suitable for scheduling. For example, it can expand the graph by generating more nodes that can be executed in parallel. If performance analysis is needed, the *program instrument* component can insert extra operations into the application to collect the execution profile. The execution profile can be used for performance visualization and post-mortem analysis.

Layer 3: Scheduling Algorithms
This layer provides a *library of scheduling algorithms*. The scheduling tool can automatically select a proper algorithm from the library according to the features of an application which is determined by the program analyzer at layer 2. User can also manually select a scheduling algorithm through the user interface at layer 1.

Layer 4: Runtime Support
This layer implements the task scheduling and execution. The *task scheduler* executes the selected scheduling algorithm to schedule the tasks to execution. The *execution monitor* collects the runtime data for performance analysis when required.

4.2 Taxonomy

Task scheduling tools can be categorized based on different characteristics. We propose a taxonomy of scheduling tools as shown in Fig. 5. The taxonomy classifies the tools with respect to four features as following.

- **Target system**: the system for which a scheduling tool is designed, including *local-area* system or *wide-area* system.

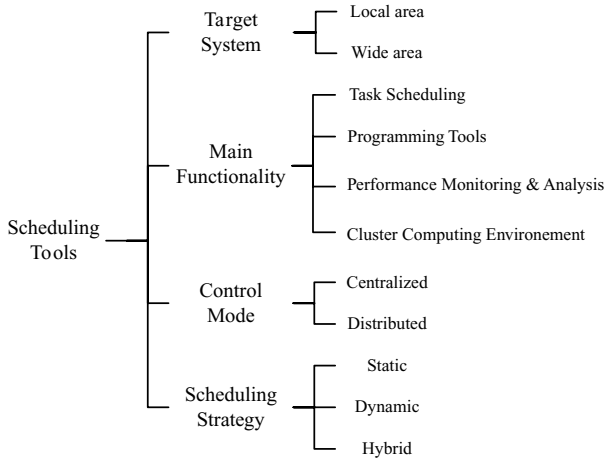


Fig. 5 Taxonomy of scheduling tools

- **Main functionality**: the main function of a scheduling tool. Some tools are dedicated for *task scheduling*. Some tools are the *programming tools* that provide the support for task scheduling. Some are *program monitoring and performance analysis tools* that evaluate the performance of scheduling algorithms. Some tools provide a comprehensive *cluster computing environment* that incorporates the support of task scheduling.
- **Control mode**: the scheduling mode adopted by a scheduling tool. *Centralized* mode is usually used for local-area systems where a centralized scheduler implements all scheduling work. *Distributed* mode is usually used for wide-area systems in which the scheduling is accomplished by the cooperation of distributed schedulers running in different system domains.
- **Scheduling policy**: the scheduling may be implemented by a *static*, *dynamic*, or *hybrid* method. As discussed in Section 3, hybrid method is a combination of static and dynamic scheduling.

The taxonomy is outlined in Fig. 5. It lists the four features used to classify the scheduling tools with possible options for each feature. For example, the scheduling tools can be classified by their main functionality. A tool can be dedicated for task scheduling. A tool can be a programming tool to support program development or even a cluster computing environment that supports task scheduling. A tool may also be designed for the performance monitoring and analysis of task scheduling algorithms and parallel programs. Section 5 will discuss different types of task scheduling tools and compare their features using this taxonomy.

5 Task scheduling tools

Various scheduling tools have been developed on parallel and distributed systems. In this section, we discuss eight representative tools.

5.1 VDCE

Virtual Distributed Computing Environment (VDCE) [55, 56] is a software development environment for building and executing large-scale applications on network of heterogeneous resources. VDCE is deployed across geographically distributed computational sites, each of which has one or more VDCE Servers. The servers provide an integrated environment of software tools and middleware for developing parallel and distributed applications meanwhile scheduling the tasks to the best available resources and managing the QoS (Quality of Service) requirements.

The VDCE software architecture consists of three parts: *Application editor*, *Application scheduler*, and *Runtime system*. Applications are developed based on a *dataflow* programming paradigm. The application editor is a web-based graphical interface for user to develop an application in the form of *application flow graph* (AFG). The editor provides menu-driven task libraries that are grouped in terms of functionality, e.g., matrix algebra library, C^3I (command, control, and communication) library, etc. For example, the AFG of a linear equation solver can be constructed using the *LU* decomposition, matrix inverse, and matrix multiplication tasks provided by the matrix algebra library.

After an AFG is created, the user can specify the properties of each task such as the computational mode (sequential or parallel), thread type (none, pthread, qthread, or cthread), communication library (P4, socket, MPI, DSM, NCS, or PVM), communication protocol (TCP/IP or ATM), system domain, cluster, machine type, and the number of processors. Then, the AFG is submitted for execution with the support of the application scheduler and the runtime system.

VDCE provides a distributed scheduling method for wide-area systems. In such a system, each site consists of a local application scheduler running on the VDCE server. The scheduling of an application is performed by the cooperation of local site and a set of remote sites. The application scheduler interprets the application flow graph and allocates the currently best available resources to the tasks. The application scheduler runs two built-in algorithms for task mapping: the *site scheduler algorithm* selects a subset of remote sites and the *host selection algorithm* at a remote site determines the best available machine in the site that can minimize the predicted execution time of each task. Then every site sends the task-to-machine mapping (i.e., machine name with predicted execution time) back to the local site. The local-site scheduler algorithm finally selects the best site based on the minimal summation of the predicted execution time and the network transfer time. The scheduling heuristic is based on the static list scheduling using *b-level* priority. The execution time of a task on a base processor has already been measured and stored in the *task-performance database* in the site repository.

The VDCE runtime system sets up the execution environment for a given application and manages its execution to meet the hardware and software requirements. The runtime system periodically measures the loads on the resources and monitors the possible failures of the resources. It also supports low latency and high-speed communication and synchronization services as well as I/O and application visualization (real-time or post-mortem visualization) services.

5.2 PYRROS

PYRROS is a software system for automatic scheduling and code generation [22, 59]. It processes an input parallel program as tasks with precedence constraints and produces code for message-passing architectures such as nCUBE-2 and INTEL-2. *Macro dataflow graph* (i.e. DAG) is used to represent a parallel program. PYRROS uses clustering algorithm to schedule a program onto parallel computers. PYRROS provides the following components:

- *Task graph language* with an interface to C or FORTRAN, which allows user to define partitioned programs and data.
- *Scheduling system* that performs the clustering of graph nodes, cluster-to-processor mapping, load balancing, and communication/computation ordering.
- *Graphic displayer* that displays task graphs and scheduling results.
- *Code generator* that inserts synchronization primitives and performs code optimization for some supercomputers.

PYRROS uses a multistage scheduling approach to schedule a DAG onto p processors in four steps:

1. *Clustering*: The tasks in a graph are grouped into clusters. The tasks in the same cluster will be assigned to one processor. PYRROS uses the Dominant Sequence Algorithm (DSC) to automatically determine the clustering of the nodes in a graph. DSC performs a sequence of clustering refinement steps. In each step, it tries to zero an edge to reduce the parallel time, i.e., the longest path called Dominant Sequence in the graph (schedule length).
2. *Cluster merging*: If the number of clusters exceeds the number of processors p, the clusters are further merged into p completely connected clusters.
3. *Physical mapping*: Maps the clusters to the physical processors. A heuristic algorithm is used for the mapping that minimizes the total communication time among the processors. The algorithm starts from an initial assignment

and performs a series of pair wise interchanges for the mapping so as to reduce the communication time. Yang and Gerasoulis [59] explained the physical mapping using the example shown in Fig. 3. Assume that the weight of each edge equals 3 time units. The communication costs between the four clusters are shown in Fig. 3(b). If four physical processors are linked as a hypercube, the clusters are optimally mapped to the processors as shown in Fig. 3(c). The clusters with the highest communication are mapped to the neighboring processors so that the total communication time can be minimized.

4. *Task ordering*: Order the execution of the tasks within each processor to minimize the total parallel time. RCP (ready critical path) algorithm is used for the ordering. The algorithm computes the *b-level* priority of each task. A task is ready if the data sent from its predecessors have arrived to it. Each processor maintains a ready priority list of tasks. The ready task with the highest priority is executed as soon as the processor becomes available. With the ready list scheduling in each processor, the total execution time can be minimized.

PYRROS was experimented on nCUBE-2 to test the performance by scheduling the LINPACK BLAS-3 based program of linear algebraic system [17].

5.3 Hypertool

Hypertool is a programming aid for automatic scheduling and synchronization on message-passing systems [58]. Hypertool takes user partitioned program as input, automatically allocating these partitions to PEs and inserting proper synchronization primitives where needed. It also produces performance estimates and quality measures for the parallel code.

User programs are defined in a uniform structure that is a sequential program with a set of procedures. Hypertool converts the program into a parallel program for a message-passing target machine by means of parallel code synthesis and optimization. It provides the lexer and parser to recognize the data dependencies and user defined partitions in a program. The graph generation module produces a *macro dataflow graph* for the program. The scheduling module assigns the graph nodes to the computing tasks for minimizing the execution time of the graph. Hypertool uses two list scheduling algorithms: (1) *Modified critical-path* (MCP) algorithm is used for the graph scheduling on a given number of PEs; (2) *Mobility-directed* (MD) algorithm is used for the scheduling on an unbound number of PEs, which chooses the optimal number of PEs that can achieve the minimal execution time. Then, the mapping module maps each computing task to a physical PE in a given topology that can minimize the network traffic. The mapping is realized by a heuristic algorithm that generates an initial assignment and then iteratively refines it to reach a better solution. After the scheduling and mapping, the synchronization module inserts the communication primitives (send and receive) to the nodes that are assigned to different PEs. Finally, the code generator generates target machine code for each PE.

Hypertool was tested on a Sun workstation to generate the scheduling of sample programs for multi-processors. The sample programs include the Gaussian elimination algorithm for solving linear systems and the Gauss-Seidel algorithm for solving Laplace equations.

5.4 CASCH

CASCH (Computer Aided SCHeduling) is a software tool for parallelizing and scheduling applications on message-passing multiprocessors [2, 35]. It was originally designed to evaluate various scheduling and mapping algorithms using the task graphs that were generated randomly, interactively, or directly from real programs.

CASCH can automatically parallelize a sequential program and add the functions of scheduling, mapping, communication, and synchronization to the parallelized program. User can write a *sequential C program* as input through a window-based interactive GUI. The structure of a user program is similar to an input program in Hypertool, i.e., a set of functions called by a main program. Communications are invoked only at the beginning and the end of a function. The *lexical analyzer* and *parser* analyze the data dependencies and the partitions of the program. The *DAG generator* generates a *macro dataflow graph* (i.e. DAG) directly from the main program with regard to the data dependencies between the functions. Each node in the graph represents a function. The weight of a node is the execution time of the function. An edge represents a message sent from one function to another. The weight of the edge is the transmission time of the message. The weights of the nodes and edges are calculated based on a database that stores the timing of various computation and communication operations on different machines. Parallel code is generated by inserting communication primitives (send, receive, etc.) into the functions.

CASCH provides a library of static list scheduling algorithms which contains three classes of algorithms according to the heuristics made: UNC (unbounded number of clusters), BNP (bounded number of processors) and APN (arbitrary processor network).

The GUI provides a graph editor to edit and display the DAGs and the target system architecture (i.e., processors and network topology). The scheduling trace can be displayed in a Gantt chart showing the start and finish time of the tasks on the processors.

CASCH has been tested by running a set of benchmarking graphs including Peer-set graphs, random graphs, and traced

graphs to evaluate the performance of different scheduling algorithms. The FFT, PDE solver, and *N*-body problem are also used to test the scheduling algorithms on a SUN workstation connected to an Intel Paragon and an IBM SP2.

5.5 ATME

ATME (Adaptive Task Mapping Environment) [28] is an environment that generates an adaptive scheduling policy as the response to the changes of the computation time of the tasks and the communication requirement as well as the change of the execution flow. ATME monitors the execution of a program and generates an adaptive scheduling for the program based on the accurate information collected from past execution profiles.

ATME provides a runtime library of process control and message passing operations for parallel programming. It accepts a parallel application in the form of DAG with the specification of the target machine topology. Before the execution, the user tasks are preprocessed into ATME tasks which in turn are analyzed and instrumented with the primitives to probe and collect the execution profiles such as computation time, communication volume, and task precedence. If a program is executed more than once, the execution profiles are aggregated and dumped into "trace files" after each execution. The trace files are maintained in the program databases. The execution profiles can be used to estimate the correspondent performance values for the next execution.

Based on a deterministic list scheduling algorithm called ERT (Earliest Ready Task) [37], ATME is able to support task scheduling when the task weights and precedence vary between executions. In the first few runs, the default scheduling method is used. When the accurate execution profiles have been collected, ATME can produce an efficient scheduling policy for later executions.

5.6 MARS

MARS (Metacomputer Adaptive Runtime System) is a framework for minimizing the execution time of distributed applications on heterogeneous, WAN-connected metacomputer [21]. MARS uses accumulated statistic data of an application's execution to derive an improved task-to-processor mapping. It also supports load balancing and task migration based on the dynamic information of processor load and network performance.

MARS views a parallel application as being composed of program phases. The MARS runtime system contains two kinds of instances: (1) The *Monitors* gather the statistic data about CPU workload, processor utilization, network performance, program phases, and communication characteristics of the applications; (2) The *Managers* utilize the statistic data to determine the task-to-processor mapping and task migration. On each of the participating metacomputer nodes, a *Network Monitor* gathers statistic data of the CPU load and network performance. The Network Monitors periodically exchange the statistic data between the computers. A preprocessor inserts extra statements into the application code to notify the MARS runtime system about the beginning of a new program phase where the *Migration Manager* will be invoked to decide whether a task re-mapping can reduce the expected execution time. The tasks with high workload will be mapped to the computer nodes with high computing power. A set of smaller tasks can be mapped to a single node to reduce the communication overhead.

In MARS, the execution trace of a parallel program is represented as a directed graph called *Dependency Graph* which is built in each run of the program based on the communication pattern. The dependency graphs from successive executions are consolidated to determine the task migration whenever a checkpoint is reached.

MARS supports C and MPI programming. This framework can also be used in other programming environments like PVM, PARMACS, and Express. The applications such as Bitonic sort and the CG (conjugate gradient) Poisson solver for PDEs are used to test the performance of the scheduling. Other applications including database and optimization problems have also been tested.

5.7 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [7, 8] is an integrated graphical environment implemented on top of PVM for creating and running parallel programs on a network of heterogeneous computers. HeNCE provides the programmers with a graphical interface to build parallel programs as well as an environment for automating the process of designing, compiling, scheduling, executing, debugging, and analyzing parallel computation. The programmers can use the graph editor to build a parallel program by drawing a *directed graph*. The nodes in the graph represent the computational procedures or the control flows (e.g., conditional branches, loops, fans, and pipes). HeNCE also provides a textual interface for user to specify a program graph in a text form. The code of the procedures can be written in C or FORTRAN. The environment provides the facilities to edit and compile the procedures on various architectures of a user-defined collection of computers called *virtual machine*. This capability allows user to specify multiple implementations of the procedures for different architectures.

To define the execution cost, a programmer can specify a *cost matrix* showing the relative costs of running the procedures on various architectures. HeNCE will automatically schedule the procedures onto particular machines based on the program graph and the cost matrix. The cost matrix is also used as an indication of machine load. The sum of all

procedure costs on a machine is viewed as the machine load. HeNCE decides the least costly placement of the nodes onto the machines using the cost matrix.

HeNCE also provides performance visualization and analysis tools. When a program is running, HeNCE can graphically display an animated view of the program state based on the program graph. Various statistics are recorded during the execution. A post-mortem performance analysis tool is associated with the graphical interface for understanding the execution flow and processor utilization.

5.8 Legion

Legion is an object-based metasystem software project. It is designed for a system composed of millions of hosts and trillions of objects tied together with high-speed networks [26, 27, 38]. Users working separately on own machines perceive an illusion of a single computer; meanwhile they can access all data and physical resources across the system. Legion builds a metacomputing system by means of transparent scheduling, data management, fault tolerance, site autonomy, and a wide range of security options.

Legion is implemented in Mentat [25], an object-oriented parallel processing system designed to simplify the programming of portable parallel applications. Mentat has two primary components: the Mentat Programming Language (MPL) and the Mentat run-time system (RTS). The MPL is an object-oriented programming language based on C++. Mentat supports both task parallelism and data parallelism. It operates over a wide spectrum of architectures from loosely-coupled heterogeneous networks of workstations to tightly-coupled multicomputers. Mentat supports medium to coarse grained applications.

The *Macro Dataflow* (MDF) model is used in Mentat and implemented by the run-time system. Macro dataflow is a medium-grained, data-driven computational model. The granularity is in the range of thousands instructions. Programs in the MDF model are represented as *directed graphs*. The Mentat compiler generates the code to construct a macro dataflow graph based on the data dependency detected at run-time. The MPL programs are executed on a virtual macro dataflow machine implemented by the Mentat run-time system.

The Mentat objects are scheduled to the processors with the purpose of minimizing the total execution time of an application. Scheduling decisions are made by a distributed algorithm which consists of two sub-decisions: (1) the *transfer policy* determines whether to process a task locally or remotely; (2) the *location policy* determines the computer node to which a task should be sent. The transfer policy is a threshold policy. Each computer node determines the scheduling based on local state information. A task originated on a node is accepted for local processing if the local state is below a threshold. Otherwise, the location policy is invoked.

Legion supports the MPL and the Basic FORTRAN Support (BFS). The BFS provides a set of Legion directives embedded in FORTRAN code. It also provides a core PVM interface and a core MPI interface to enable the PVM and MPI applications to use Legion features. Legion is designed to support a wide range of massively parallel applications such as CFD computations, climate and ocean modeling and simulation.

5.9 Comparison of the tools

The scheduling tools discussed above can be compared using the taxonomy defined in Section 4.2. Table 1 summarizes the features of these tools using the characteristics given in the taxonomy (i.e., target system, main functionality, control mode, and scheduling policy) as well as four additional features as following:

- **Task graph**: the form of task graph. Most of the tools use directed acyclic graph (DAG). Some tools use a special form of DAG.
- **Scheduling algorithm**: the scheduling algorithm(s) used by a tool such as list scheduling, clustering, and adaptive scheduling which decides a schedule based on the execution profile of an application.
- **User interface**: a tool may provide some kind of user interface such as program/graph editor, program and performance visualization.
- **Programming paradigm**: the programming languages and libraries supported by the tool.

As Table 1 shows, these scheduling tools are developed for different purposes. The earlier systems like PYRROS and Hypertool are designed for the task scheduling on message-passing parallel machines. VDCE and HeNCE are software development environments for supporting parallel programming and task scheduling on heterogeneous networks. Legion is a more powerful metasystem that provides comprehensive supports including resource management, data management, program development, fault tolerance, and security for wide-area systems. MARS is a performance monitoring and task scheduling tool for metacomputer, which collects the execution profile of a program and performs adaptive task-to-processor mapping based on the profile. ATME is also a tool for adaptive task mapping based on execution profile. CASCH is a tool to evaluate the performance of list scheduling algorithms, which also supports automatic parallelization of serial programs.

From Table 1, we can find that the task scheduling on local-area systems is usually implemented in a centralized mode where a single scheduler is running on one of the hosts such as in PYRROS, Hypertool, CASCH, ATME, and HeNCE.

Table 1 Comparison of scheduling tools

	VDCE	PYRROS	Hypertool	CASCH	ATME	MARS	HeNCE	Legion
Target system	Wide-area network of heterogeneous computers	Message passing parallel machines	Message passing parallel systems	Message passing multiprocessors	Parallel systems	Heterogeneous WAN-connected metacomputers	Network of heterogeneous computers	Wide-area systems
Main functionality	Software development environment and performance or workload visualization tool	Automatic task scheduling and code generation	Automatic scheduling, synchronization, code generation and mapping	Automatic code parallelization, scheduling and performance evaluation tool for scheduling algorithms	Adaptive task mapping environment and post-mortem analysis tool	Adaptive task mapping tool	Software development environment	Metacomputing system
Control mode	Distributed	Centralized	Centralized	Centralized	Centralized	Distributed	Centralized	Distributed
Scheduling policy	Static	Static	Static	Static	Static	Hybrid	Dynamic	Dynamic
Task graph	DAG: Application flow graph	DAG: Macro dataflow graph	DAG: Macro dataflow graph	DAG: Macro dataflow graph	DAG	DAG: Dependency graph	Directed graph with special nodes for control flow	Macro dataflow graph
Scheduling algorithm	List scheduling	Clustering	List scheduling	A library of list scheduling algorithms	Adaptive scheduling	Adaptive scheduling	Graph-based scheduling	Macro dataflow graph based scheduling
User interface	Web-based GUI to edit application graph and performance visualization	GUI to input user program and display task graph or scheduling results	Interface to input user program	GUI to edit or display user program, task graph, system architecture, and performance visualization	Unknown	Unknown	GUI for program development and performance analysis	Legion Grid Portal
Programming paradigm	Proprietary task libraries, P4, PVM, MPI, etc.	C, FORTRAN	Unknown	C	C, PVM	C, MPI, PVM, etc.	C, FORTRAN, PVM	MPL, BFS, PVM, MPI

Since distributed system is expanding to wide-area environment, distributed scheduling strategy is required to satisfy the resource heterogeneity, site autonomy and fault-tolerance requirements. The tools for wide-area systems such as VDCE, MARS and Legion implement task scheduling based on a distributed mode.

Most of the tools use a form of DAG to present the applications for scheduling. The deterministic structure of the DAG facilitates the design and implementation of the scheduling algorithms. However, DAG lacks the ability to describe complex program structures such as iteration and changeable communication pattern. Therefore, alternative forms of task graph are required to enhance the capability of representing various application structures such as the directed graph in HeNCE that includes special nodes to represent control flow.

These tools are not originally designed for cluster computing. However, they can be applied to cluster systems. In principle, the tools designed for local-area systems including PYRROS, Hypertool, CASCH, ATM and HeNCE can be directly used on clusters. For the tools developed for wide-area systems including VDCE, MARS and Legion, a cluster can be viewed as a special case of the system that contains only one local site. The local-site scheduling algorithms in these tools can be used on the cluster system. As the rapid advance of cluster computing, however, these tools do not fully suit the architectural and application requirements of cluster systems. New scheduling tools need to be developed with the technological improvements as discussed in the next section.

6 Discussion and conclusions

This paper explores the application scheduling approaches and tools for cluster computing. The scheduling techniques are important to realize high-performance parallel computing. Task graph, commonly represented as a weighted *directed acyclic graph*, is a general model to represent a parallel program for task scheduling. Different scheduling algorithms have been proposed based on various heuristics on program features and system architecture. The task graph based scheduling has been considered as an effective model for the scheduling algorithms in theoretical studies. However, it has not obtained wide use in practice. Few task scheduling tools can be found on cluster systems. So far, no commercial tool is available.

The usability of the scheduling techniques is restricted by various factors. One factor is the discrepancy between the simplified DAG structure and the complex real application structures. DAG can only describe non-iterative computations in a straightforward way. It is not a model feasible to describe complex program structures. As applications usually contain loops and branches, more complex graph structures should be adopted to describe these structures. On the other hand, complex graphs may make the task scheduling intractable in acceptable time complexity.

Another drawback of DAG is the low scalability. The topology of a DAG is highly related to the problem size of an application and the number of processors in use. When the problem size or the number of processors has changed, a new task graph should be drawn and a new scheduling should be determined for it. In DAG, the iteration has to be unrolled. An unrolled graph is usually oversized beyond the scope of graph drawing and display facility. The low scalability discourages the real use of the DAG-based scheduling approaches.

To improve the usability of the graph based scheduling, the method should be improved in the following aspects.

(1) *Task graph model*
A task graph should be highly scalable to the program structure. In other words, a task graph ought to represent the logical structure of a program which is independent from the problem size and the number of processors. To support the scalability, adaptive *graph transformation* is needed to adapt a graph to these parameters when task scheduling is being conducted. The clustering approach discussed in Section 3.2.3 can be used to merge the graph nodes when the number of parallel tasks exceeds the available processors. The task graph should also allow graph expansion to match the increased problem size or the number of processors by decomposing the graph nodes and reconstructing the edges. In the framework shown in Fig. 4, the graph transformation component in layer 2 is defined for this purpose.

(2) *Scheduling strategy*
For a wide-area system, high autonomy, heterogeneity and scalability are the key merits of a scheduling strategy. A distributed scheduling strategy is suited to wide-area environment in which the scheduling of an application is accomplished by the cooperation of the schedulers in distributed domains. The scheduling algorithms need to consider the network delay and the heterogeneous computing resources so as to fully utilize the computing power and reduce the communication overhead. In the framework shown in Fig. 4, distributed scheduling algorithms should be included in the library of scheduling algorithms in layer 3 to satisfy the scheduling requirements in wide-area systems.

With all these efforts, the task graph based scheduling can be expected to acquire broad use in high-performance cluster computing.

References

1. I. Ahmad and Y. Kwok, On exploiting task duplication in parallel program scheduling. *IEEE Trans. on Parallel and Distributed Systems* 9(9) (September 1998) 872–892.
2. I. Ahmad, Y. Kwok, M. Yu, and W. Shu, CASCH: a software tool for automatic parallelization and scheduling of programs on message-passing multiprocessors. IEEE Concurrency 8(4) (October–December 2000) 21–33.
3. G. Aloisio and M. Bochicchio, The use of PVM with workstation clusters for distributed SAR data processing, in: *Proceedings of HPCN Europe 1995*, Milan, Italy, LNCS 919, Springer, (May 1995) pp. 570–581.
4. O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo, A comparative study of online scheduling algorithms for networks of workstations, Cluster Computing 3(2) (2000) 95–112.
5. M. Baker, G. Fox, and H. Yau, Cluster management software (1996) http://www.crpc.rice.edu/NHSEreview/CMS/.
6. M. Baker, (ed.), Cluster computing white paper, IEEE Computer Society Task Force on Cluster Computing (TFCC) http://www.csm.port.ac.uk/∼mab/tfcc/WhitePaper/ (December 2000).
7. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, and V. Sunderam, HeNCE: graphical development tools for network-based concurrent computers, in: *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, IEEE Computer Society Press, (April 1992) pp. 129–136.
8. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore, HeNCE: a heterogeneous network computing environment, Scientific Programming 3(1) (1994) 49–60.
9. J. Bernabeu, Y. Khalidi, V. Matena, K. Shirriff, and M. Thadani, Solaris MC: a multi-computer OS, Technical Report: TR-95-48, Sun Microsystems http://research.sun.com/research/techrep/1995/abstract-48.html.
10. R. Buyya (ed), *High Performance Cluster Computing: Architectures and Systems*, vol. 1, (Prentice Hall PTR, NJ, USA 1999).
11. R. Buyya, (ed), *High performance cluster computing: programming and applications*, vol. 2, Prentice Hall PTR, NJ, USA (1999).
12. J. Cao, A. Chan, Y. Sun, and K. Zhang, Dynamic Configuration Management in Graph-Oriented Distributed Programming Environment, Science of Computer Programming 48(1) (July 2003) 43–65.
13. Condor, http://www.cs.wisc.edu/condor/.
14. M. Cosnard and E. Jeannot, Compact DAG representation and its dynamic scheduling, Journal of Parallel and Distributed Computing 58(3) (September 1999) 487–514.
15. S. Darbha and D. Agrawal, A fast and scalable scheduling algorithm for distributed memory systems, in: *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, San Antonio, TX (October 25–28, 1995) pp. 60–63.
16. S. Darbha and D. Agrawal, A task duplication based scalable scheduling algorithm for distributed memory systems, Journal of Parallel and Distributed Systems 46(1) (October 1997) 15–27.
17. J. Dongarra, J. Croz, I. Duff, and S. Hammarling, A set of level 3 basic linear algebra subprograms, ACM Trans. on Mathematical Software 16(1) (1990) 1–17.
18. D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, Theory and practice in parallel job scheduling, in: *Proceedings of 3rd Workshop on Job Scheduling Strategies for Parallel Processing* , LNCS 1291, Springer-Verlag, (1997) pp. 1–34.
19. D. Feitelson, A survey of scheduling in multiprogrammed parallel systems, Research Report RC 19790 (87657), IBM T. J. Watson Research Center (October 1994), Revised version in August 1997.
20. D. Feitelson, Scheduling parallel jobs on clusters, in: *High Performance Cluster Computing: Architectures and Systems*, vol. 1, Rajkumar Buyya (ed.), Prentice-Hall (1999) pp. 519–533.
21. J. Gehring and A. Reinefeld, MARS-a framework for minimizing the job execution time in a metacomputing environment, Future Generation Computer Systems 12(1) (1996) 87–99.
22. A. Gerasoulis, J. Jiao, and T. Yang, A multistage approach to scheduling task graphs, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 22 (1995) 81–105.
23. GLUnix, http://now.cs.berkeley.edu/Glunix/glunix.html.
24. A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed., Pearson Education (2003).
25. A. Grimshaw, J. Weissman, and W. Strayer, Portable run-time support for dynamic object-oriented parallel processing, ACM Trans. on Computer Systems 14(2) (May 1996) 139–170.
26. A. Grimshaw, W. Wulf, and the Legion team, The Legion vision of a worldwide virtual computer, Communications of ACM 40(1) (January 1997) 39–45.
27. A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, Legion: an operating system for wide-area computing, IEEE Computer 32(5) (May 1999) 29–37.
28. L. Huang, M. Oudshoorn and J. Cao, Design and implementation of an adaptive task mapping environment for parallel programming, Australian Computer Science Communications 19(1) (February 1997) 326–335.
29. K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture*, *Programming*, *(WCB/McGraw-Hill* 1998).
30. IBM Redbook, Workload management with LoadLeveler, http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246038.html?Open.
31. IBM@server Cluster 1600 http://www-1.ibm.com/servers/eserver/clusters/hardware/1600. html.
32. H. James, Scheduling in metacomputing systems, Ph.D. thesis, University of Philosophy (July 1999) http://www.dhpc.adelaide.edu.au/reports/057/html.
33. C. Jenssen, Parallel computational fluid dynamics 2000: trends and applications, Elsevier Science (2001).
34. J. Krallmann, U. Schwiegelshohn, and R. Yahyapour, On the design and evaluation of job scheduling algorithms, in: *Proce of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, in conjunction with IPPS/SPDP'99, San Juan, Puerto Rico (April 16, 1999) pp. 17–42.
35. Y. Kwok and I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, Journal of Parallel and Distributed Computing 59(3) (December 1999) 381–422.
36. Y. Kwok and I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys 31(4) (December 1999) 406–471.
37. C. Lee, J. Hwang, Y. Chow, and F. Anger, Multiprocessor scheduling with interprocessor communication delays, Operations Research Letters 7(3) (June 1988) 141–147.
38. Legion, http://legion.virginia.edu/.
39. LSF, http://www.platform.com/products/wm/LSF/index.asp.
40. MOSIX, http://www.mosix.com/.
41. MPI, http://www-unix.mcs.anl.gov/mpi/.
42. Myricom: Creator of Myrinet, http://www.myri.com/.
43. OpenMP, http://www.openmp.org/.
44. G. F. Pfister, In Search of Clusters, 2nd ed., (Prentice Hall 1998).
45. PVM, http://www.csm.ornl.gov/pvm/pvm_home.html.
46. S. Ranaweera and D. Agrawal, A task duplication based scheduling algorithm for heterogeneous systems, in: *Proceedings of 14th International Parallel and Distributed Processing Symposium* (IPDPS'2000), Cancun, Mexico (May 1–5, 2000) pp. 445–450.

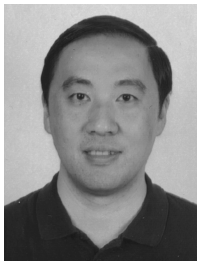
47. H. El-Rewini, T. Lewis, and H. Ali, Task Scheduling in Parallel and Distributed Systems, (Prentice Hall PTR, NJ 1994).
48. M. Senar, A. Ripoll, A. Cortes, and E. Luque, Clustering and reassignment-based mapping strategy for message-passing architectures, in: *Proceedings of IPPS/SPDP 1998, Orlando, Florida* (March 30-April 3, 1998) pp. 415–421.
49. H. Shen, S. Lor, and P. Maheshwari, An architecture-independent graphical tool for automatic contention-free process-to-processor mapping, The Journal of Supercomputing 18(2) (February 2001) 115–139.
50. S. Spach and R. Pulleyblank, Parallel raytraced image generation, Hewlett-Packard Journal 43(3) (June 1992) 76–83.
51. J. Squyres, A. Lumsdaine, and R. Stevenson, A cluster-based parallel image processing toolkit, Visual Data Exploration and Analysis III, vol. 2421 of SPIE Proceedings, Society of Photo-optical Instrumentation Engineers (SPIE) (1995) pp. 228–239.
52. Sun Microsystems, Sun[tm] Clusters: providing enterprise-wide business-critical computing, White Paper (October 1997) http://wwws.sun.com/software/cluster/wp-sunclusters/.
53. Y. Sun and C. Wang, Solving irregularly structured problems based on distributed object model, Parallel Computing 29(11/12) (November 2003) 1539–1562.
54. T. Tamura, M. Oguchi, and M. Kitsuregawa, Parallel database processing on a 100 node PC cluster: cases for decision support query processing and data mining, in: *Proceedings of Supercomputing Conference* (SC'97), San Jose (November 15–21, 1997).
55. H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing, and B. Ye, The software architecture of a virtual distributed computing environment, in: *Proceedings of 6th International Symposium on High Performance Distributed Computing (HPDC'97)*, Portland, OR (August 5–8, 1997) pp. 40–49.
56. H. Topcuoglu, S. Hariri, D. Kim, Y. Kim, X. Bing, B. Ye, I. Ra, and J Valente, The design and evaluation of a virtual distributed computing environment, Cluster Computing 1(1) (1998) 81–93.
57. Windows 2000 Cluster Technologies, http://www.microsoft.com/windows2000/technologies/clustering/
58. M. Wu and D. Gajski, Hypertool: a programming aid for message-passing systems, IEEE Trans. on Parallel and Distributed Systems 3(1) (1990) 330–343.
59. T. Yang and A. Gerasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, in: *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington D.C. (1992) pp. 428–437.

Jiannong Cao received the BSc degree in computer science from Nanjing University, Nanjing, China in 1982, and the MSc and the Ph.D degrees in computer science from Washington State University, Pullman, WA, USA, in 1986 and 1990 respectively.

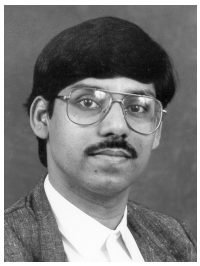
He is currently an associate professor in Department of Computing at the Hong Kong Polytechnic University, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. He was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile computing, fault tolerance, and distributed software architecture and tools. He has published over 120 technical papers in the above areas. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/programme committee member for many international conferences.

Dr. Cao is a member of the IEEE Computer Society, the IEEE Communication Society, IEEE, and ACM. He is also a member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing, and Computer Architecture Professional Committee of the China Computer Federation.

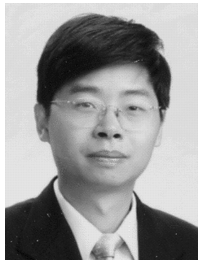
Alvin Chan is currently an assistant professor at the Hong Kong Polytechnic University. He graduated from the University of New South Wales with a Ph.D. degree in 1995 and was subsequently employed as a Research Scientist by the CSIRO, Australia. From 1997 to 1998, he was employed by the Centre for Wireless Communications, National University of Singapore as a Program Manager. Dr. Chan is one of the founding members and director of a university spin-off company, Information Access Technology Limited. He is an active consultant and has been providing consultancy services to both local and overseas companies. His research interests include mobile computing, context-aware computing and smart card applications.



Yudong Sun received the B.S. and M.S. degrees from Shanghai Jiao Tong University, China. He received Ph.D. degree from the University of Hong Kong in 2002, all in computer science. From 1988 to 1996, he was among the teaching staff in Department of Computer Science and Engineering at Shanghai Jiao Tong University. From 2002 to 2003, he held a research position at the Hong Kong Polytechnic University. At present, he is a Research Associate in School of Computing Science at University of Newcastle upon Tyne, UK. His research interests include parallel and distributed computing, Web services, Grid computing, and bioinformatics.



Sajal K. Das is currently a Professor of Computer Science and Engineering and the Founding Director of the Center for Research in Wireless Mobility and Networking (CReWMaN) at the University of Texas at Arlington. His current research interests include resource and mobility management in wireless networks, mobile and pervasive computing, sensor networks, mobile internet, parallel processing, and grid computing. He has published over 250 research papers, and holds four US patents in wireless mobile networks. He received the Best Paper Awards in ACM MobiCom'99, ICOIN-16, ACM, MSWiM'00 and ACM/IEEE PADS'97. ; Dr. Das serves on the Editorial Boards of IEEE Transactions on Mobile Computing, ACM/Kluwer Wireless Networks, Parallel Processing Letters, Journal of Parallel Algorithms and Applications. He served as General Chair of IEEE PerCom'04, IWDC'04, MASCOTS'02 ACM WoWMoM'00-02; General Vice Chair of IEEE PerCom'03, ACM MobiCom'00 and IEEE HiPC'00-01; Program Chair of IWDC'02, WoWMoM'98-99; TPC Vice Chair of ICPADS'02; and as TPC member of numerous IEEE and ACM conferences.



Minyi Guo received his Ph.D. degree in information science from University of Tsukuba, Japan in 1998. From 1998 to 2000, Dr. Guo had been a research scientist of NEC Soft, Ltd. Japan. He is currently a professor at the Department of Computer Software, The University of Aizu, Japan. From 2001 to 2003, he was a visiting professor of Georgia State University, USA, Hong Kong Polytechnic University, Hong Kong. Dr. Guo has served as general chair, program committee or organizing committee chair for many international conferences, and delivered more than 20 invited talks in USA, Australia, China, and Japan. He is the editor-in-chief of the Journal of Embedded Systems. He is also in editorial board of International Journal of High Performance Computing and Networking, Journal of Embedded Computing, Journal of Parallel and Distributed Scientific and Engineering Computing, and International Journal of Computer and Applications.

Dr. Guo's research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, data mining, molecular computing and software engineering. He is a member of the ACM, IEEE, IEEE Computer Society, and IEICE. He is listed in Marquis Who's Who in Science and Engineering.