■■■■■ CHAPTER 6

Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach

MITCHELL D. THEYS

University of Illinois at Chicago

TRACY D. BRAUN, HOWARD JAY SIEGEL, and ANTHONY A. MACIEJEWSKI,

Purdue University

YU-KWONG KWOK

The University of Hong Kong

6.1   INTRODUCTION

Different portions of an application task often require different types of computation. In general, it is impossible for a single machine architecture with its associated compiler, operating system, and programming tools to satisfy all the computational requirements in such an application equally well. One type of *heterogeneous computing* (*HC*) environment consists of a suite of different types of machines, high-speed interconnections, interfaces, operating systems, communication protocols, and programming environments that provide a variety of architectural capabilities. Such an HC environment can be orchestrated to perform an application that has diverse execution requirements [17, 20, 22, 32, 46, 47, 50]. An application task can be decomposed into *subtasks*, where each subtask is computationally homogeneous (well-suited to a single machine), and different subtasks may have different machine architectural requirements. These subtasks may have data dependencies among them. A group of independent tasks, known as a *meta-task*, can also be executed in the HC environment. The tasks in a *meta-task* have no data dependencies among them, and may have different architectural requirements.

Once the set of subtasks or independent tasks to be executed is known, the following decisions must be made: *matching*, i.e., assigning subtasks or independent tasks to machines, and *scheduling*, i.e., ordering subtask or independent task execution for each machine and (for subtasks) ordering intermachine data transfers. The resulting matching and scheduling scheme is called a *mapping*. In the studies reported here, the primary goal of HC is to find a mapping that will achieve the minimal *completion time*, i.e., the minimal overall execution time of the application task or meta-task on the machine suite.

It is well known that such a mapping problem is, in general, NP complete [19, 29]. A number of approaches to different aspects of this problem have been proposed (e.g., [18, 20, 30, 38, 51, 55]). These approaches include both static and dynamic heuristics. *Static* heuristics are executed off-line, in a planning mode, when the application task or meta-task to map is known in advance. *Dynamic* heuristics are executed on-line, in real-time, and can make use of feedback from the HC system (e.g., [37]). Heuristics developed to perform the mapping function are often difficult to compare because of different underlying assumptions in the original studies of each heuristic (e.g., number and types of machines in the HC suite) [5]. Various researchers have considered the use of genetic-algorithm-based approaches for this mapping function (e.g., [45, 48, 52, 61]). This chapter summarizes three particular genetic-algorithm-based approaches, each for a different HC environment.

Two static approaches and one dynamic approach used for solving the mapping problems for different HC environments are discussed. The first approach decides the subtask to machine assignments, orders the execution of the subtasks assigned to each machine, and schedules the data transfers among subtasks [58]. The characteristics of this approach include separation of the matching and the scheduling representations, independence of the chromosome structure from the details of the communication subsystem, and consideration of overlap among all computations and communications that obey subtask precedence constraints. The computation and communication overlap is limited only by intersubtask data dependencies and machine/network availability. This genetic-algorithm-based approach can be applied to performing the mapping in a variety of HC systems. It is applicable to the static mapping of production jobs and can be readily used to collectively schedule a set of tasks that are decomposed into subtasks.

The second approach focuses on a particular application domain [*iterative automatic target recognition (ATR)* tasks] and an associated specific class of dedicated heterogeneous parallel hardware platforms. For the computational environment considered, a methodology for real-time, on-line, input-data-dependent remapping of the application subtasks to the processors (machines) in the heterogeneous parallel hardware platform using a previously stored off-line, statically determined, mapping is presented [8, 34, 35]. The operating system makes a heuristic-based decision during the execution of the application whether to perform a remapping based on information generated by the application from its input data. If the decision is to remap, the operating system selects a previously determined and stored genetic-algorithm-based mapping that is appropriate for the given state of the application (e.g., the number of objects it is currently tracking).

The third approach examines the mapping of meta-tasks to machines in an HC suite. It is assumed in this approach that each machine executes a single task at a time, in the order in which the tasks arrive, and there are no dependencies among the tasks. Because of these assumptions, scheduling is simplified and the resulting solutions of the mapping heuristics focus more on finding an efficient matching of tasks to machines. Eleven different static mapping heuristics from the literature were implemented and compared by simulation studies under one set of common assumptions [6]. One of these heuristics was a genetic algorithm, providing insight into the relative performance of genetic algorithms to other mapping techniques.

The organization of this chapter is as follows. The details of the problem and assumptions made for each of the approaches are presented in Section 6.2. Section 6.3 briefly describes the basic steps of any genetic algorithm. In Section 6.4, a summary of the results from the static subtask mapping approach is presented [58]. Section 6.5 discusses results obtained from the on-line use of off-line-derived mappings approach [8, 34]. Finally, Section 6.6 details experiences from implementing the static meta-task mapping approach [6].

The research reported in this chapter was supported in part by the DARPA/ITO Quorum Program project called *MSHN* (Management System for Heterogeneous Networks) [26]. MSHN is a collaborative research effort among the Naval Postgraduate School, NOEMIX, Purdue University, and the University of Southern California. The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver the requested quality of service. The heuristics developed in this chapter or their derivatives may be included in the MSHN prototype.

6.2 PROBLEM DESCRIPTIONS

6.2.1 Static Matching and Scheduling of Subtasks

There are many open research problems in the HC field [36, 46]. To isolate and focus on the mapping problem, researchers typically make assumptions about other components of an overall HC system (e.g., [45, 48]). This subsection considers the assumptions made in [58] for the static mapping of subtasks.

It is assumed that the application task is written in some machine-independent language (e.g., [59]). It is also assumed that an application task is decomposed into multiple subtasks and the data dependencies among them are known and are represented by a *directed acyclic graph (DAG)*. If intermachine data transfers are data dependent, then some set of expected data transfers must be assumed. The estimated expected execution time for each subtask on each machine is assumed to be known a priori. Finding the estimated expected execution times for subtasks is another research problem, which is beyond the scope of this chapter. Approaches based on task profiling and analytical benchmarking are surveyed in [32, 46, 47]. The HC system is assumed to have operating system support for executing each

subtask on the machine it is assigned and for performing intermachine data transfers as scheduled by this genetic-algorithm-based approach.

In the type of HC system considered here, an application task is decomposed into a set of subtasks S. Define $|S|$ to be the number of subtasks in the set, and s_i to be the ith subtask $0 \leq i < |S|$. An HC environment consists of a set of machines M. Define $|M|$ to be the number of machines and m_j to be the jth machine, $0 \leq j < |M|$. The *global data items* (*gdis*), i.e., data items that need to be transferred between subtasks, form a set G. Define $|G|$ to be the number of items and gdi_k to be the kth global data item, $0 \leq k < |G|$.

It is assumed that for each global data item there is a single subtask that produces it (*producer*), and there are some subtasks that need this data item (*consumers*). Hence, the task is represented by a *single-producer directed acyclic graph* (*SPDAG*). Each edge goes from a producer to a consumer and is labeled by the gdi that is transferred over it. Figure 6.1 shows an example SPDAG.

The following further assumptions are made for the static subtask mapping problem. One is the exclusive use of the HC environment for the application; and that the genetic-algorithm-based mapper is in control of the HC machine suite. Another is nonpreemptive subtask execution. Also, all input data items of a subtask must be received before its execution can begin, and none of its output data items are available until the execution of this subtask is finished. If a data conditional is based on input data, it is assumed to be contained inside a subtask. A loop that uses an input data item to determine one or both of its bounds is also assumed to be contained inside a subtask. These restrictions help make the mapping problem more manageable, and solving this problem under these assumptions is a significant step toward solving the general mapping problem.

For any static heuristic, it is assumed that an accurate estimate of the expected execution time for each subtask on each machine is known prior to execution and contained within an $|M| \times |S|$ ETC (expected time to compute) *matrix*. ETC(i, j) is the estimated execution time for subtask s_i on machine m_j.

These times are assumed to include the time to move the executables and initial data (not gdis) associated with subtask s_i to machine m_j when necessary.] The
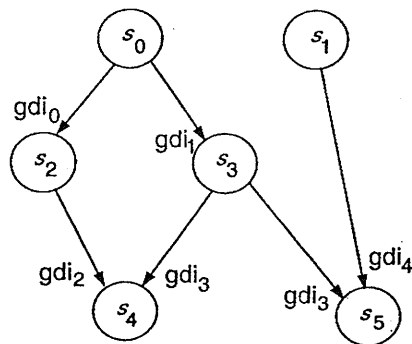

Figure 6.1 An example SPDAG.

assumption that these estimated expected execution times are known is commonly made when studying mapping heuristics for HC systems (e.g., [23, 31, 48]).

6.2.2 Semistatic Matching and Scheduling of Subtasks

In [7, 8], a detailed design of an *intelligent operating system (IOS)* is proposed for a particular application domain in which (1) an iterative application is to be mapped onto an associated specific type of dedicated heterogeneous parallel hardware platform; and (2) the execution of each iteration can be represented by a DAG of subtasks. To minimize the execution time of such an iterative application in a heterogeneous parallel computing environment, an appropriate mapping scheme is needed. However, when some of the characteristics of the application subtasks are unknown a priori and will change from iteration to iteration during execution-time, it may not be feasible or desirable to use the same off-line-derived mapping throughout the whole execution of the application.

In such situations, a semistatic methodology [7, 8, 35] can be employed that starts with an initial mapping, but dynamically decides whether to remap the application with a mapping previously determined off-line. This is done by observing, from one iteration to another, the effects of the changing characteristics of the application's input data, called *dynamic parameters*, on the application's execution time. That is, the IOS will be able to make a heuristically determined decision during the execution of the application whether to perform a remapping based on information generated by the application from its input data. If the decision is to remap, the IOS will be able to select a precomputed and stored mapping that is appropriate for the given state of the application (e.g., the number of objects the ATR system is currently tracking). This remapping process will, in general, require a certain system *reconfiguration time* for relocating the data and program modules.

The semistatic method differs considerably from other real-time HC mapping techniques in that it involves the on-line, real-time use of off-line, precomputed mappings. This is significant because it is possible for off-line heuristics to have much longer execution times to search for a good solution than what is practical for an on-line heuristic. Thus, with the semistatic method, the mapping quality of an off-line, time-consuming heuristic can be approached at real-time speeds. As detailed in Section 6.5.6, the GA described in Section 6.4 is enhanced to be used as an off-line heuristic for determining high-quality mappings for on-line use. In an extensive simulation study, it was found that the semistatic method is much more effective than a purely dynamic approach.

6.2.3 Static Matching and Scheduling for Meta-Tasks

This subsection considers the assumptions made in [6] for the static mapping of meta-tasks. Recall from Section 6.1 that a meta-task is defined as a collection of independent tasks with no data dependencies (a given task, however, may have subtasks and dependencies among the subtasks). For this case study, it is assumed that static (i.e., off-line or predictive) mapping of meta-tasks is being performed. (In

some systems, all tasks and subtasks in a meta-task, as defined earlier, are referred to as just tasks.)

It is also assumed that each machine executes a single task at a time in the order in which the tasks arrived. Because there are no dependencies among the tasks, scheduling is simplified, and thus the resulting solutions of the mapping heuristics focus more on finding an efficient matching of tasks to machines. It is also assumed that the size of the meta-task T (number of tasks to execute) is $|T|$, and the number of machines in the HC environment is $|M|$, and both are static and known a priori. Expected task execution times are specified in an $|M| \times |T|$ ETC matrix, analogous to the one defined in Section 6.2.1 for subtasks. It is assumed that the HC system is dedicated for the meta-task, and controlled by the mapper.

6.3 GENETIC ALGORITHM OVERVIEW

Genetic algorithms (GAs) are a useful heuristic approach to finding near-optimal solutions in large search spaces [14, 25, 27]. There is a large variety of approaches to GAs; many are surveyed in [40, 49]. The following is a brief overview of GAs to provide background for the description of the proposed approaches.

The first step necessary to employ a GA is to encode some of the possible solutions to the optimization problem as a set of strings *(chromosomes)*. Each chromosome represents one solution to the problem, and a set of chromosomes is referred to as a *population*. The next step is to derive an initial population. A random set of chromosomes is often used as the initial population. Some specified chromosomes can also be included as *seeds*. This initial population is the first generation from which the evolution begins.

The third step is to evaluate the quality of each chromosome. Each chromosome is associated with a *fitness value*, which in this case is the completion time of the solution (mapping) represented by this chromosome (i.e., the expected execution time of the application task or meta-task if the mapping specified by this chromosome were used). The objective of the GA search is to find a chromosome that has the optimal fitness value. The *selection* process is the next step. In this step, each chromosome is eliminated or duplicated (one or more times) based on its relative quality. The population size is typically kept constant.

Selection is followed by the *crossover* step. With some probability, pairs of chromosomes are selected from the current population and some of their corresponding components are exchanged to form two valid chromosomes, which may or may not already be in the current population. After crossover, each string in population can be *mutated* with some probability. The mutation process transforms a chromosome into another valid chromosome that may or may not already be in the current population. The new population is then evaluated. If none of the stopping criteria has been met, the new population goes through another cycle (iteration) of selection, crossover, mutation, and evaluation. These cycles continue until one of the stopping criteria is met.

```
GA_matching_scheduling {
    initial population generation;
    evaluation;
    while (stopping criteria not met) {
        selection;
        crossover;
        mutation;
        evaluation;
    }
    output the best solution found;
}
```

Figure 6.2 General procedure for a genetic algorithm, based on [49].

In summary, the following items must be determined to implement a GA for a given optimization problem: (1) an encoding, (2) an initial population, (3) an evaluation using a particular fitness function, (4) a selection mechanism, (5) a crossover mechanism, (6) a mutation mechanism, and (7) a set of stopping criteria. The outline of the GA-based approach is shown in Fig. 6.2. Details of three GA-based approaches will be discussed in the following sections.

6.4 STATIC MATCHING AND SCHEDULING OF SUBTASKS

6.4.1 Introduction

This section discusses the GA-based approach found in [58] for the static mapping of subtasks. This section presents the details about the chromosome representation used, how population generation was performed, the mutation and crossover operators used, and comparisons with nonevolutionary approaches.

6.4.2 Chromosome Representation

Each chromosome consists of two parts: the matching string and the scheduling string. Let *mat* be the *matching string*, which is a vector of length $|S|$, such that $mat(i) = m_j$, where $0 \le i < |S|$ and $0 \le j < |M|$, i.e., subtask s_i is assigned to machine m_j.

The *scheduling string (ss)* is a topological sort [12] of the SPDAG, i.e., a total ordering of the nodes (subtasks) in the SPDAG that obeys the precedence constraints. Define ss to be the scheduling string, which is a vector of length $|S|$, such that $ss(k) = s_i$, where $0 \le i, k < |S|$, and each s_i appears only once in the vector, i.e., subtask s_i is the kth subtask in the scheduling string. Because it is a topological sort, if ss(k) is a consumer of a global data item produced by ss(j), then $j < k$. The scheduling string gives an ordering of the subtasks that is used by the evaluation step.

Then in this GA-based approach, a chromosome is represented by a two-tuple [mat,ss]. Thus, a chromosome represents the subtask-to-machine assignments (matching) and the execution ordering of the subtasks assigned to the same machine. The scheduling of the global data item transfers and the relative ordering of subtasks assigned to different machines are determined by the evaluation step. Figure 6.3 illustrates two different chromosomes for the SPDAG in Fig. 6.1, for $|S| = 6$, $|M| = 3$, and $|G| = 5$.

6.4.3   Initial Population Generation

In the initial population generation step, a predefined number of chromosomes are generated, the collection of which forms the initial population. When generating a chromosome, a new matching string is obtained by randomly assigning each subtask to a machine. To form a scheduling string, the SPDAG is first topologically sorted to form a basis scheduling string. Then, for each chromosome in the initial population, this basis string is mutated a random number of times (between one and the number of subtasks) using the scheduling-string mutation operator to generate the ss vector (which is a valid topological sort of the given SPDAG). Furthermore, it is common in GA applications to incorporate solutions from some nonevolutionary heuristics into the initial population, which may reduce the time needed for finding a satisfactory solution [14]. In this GA-based approach, along with those chromosomes
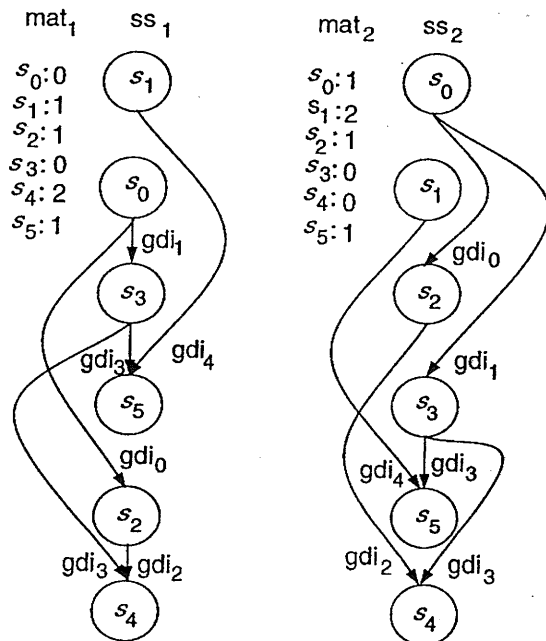

Figure 6.3   Two chromosomes from the SPDAG in Fig. 6.1.

representing randomly generated solutions, the initial population also includes a seed chromosome that represents the solution from a nonevolutionary baseline heuristic (see Section 6.4.8).

Each newly generated chromosome is checked against those previously generated. If a new chromosome is identical to any of the existing ones, it is discarded and the process of chromosome generation is repeated until a unique new chromosome is obtained.

6.4.4   Selection

In this step, the chromosomes in the population are first ordered (*ranked*) by their fitness values from the best to the worst. Those having the same fitness value are ranked arbitrarily among themselves. Then a *rank-based roulette wheel selection scheme* is used to implement the selection step [27, 49]. In the rank-based selection scheme, each chromosome is allocated a sector on a roulette wheel. Let P denote the population size and A_i denote the angle of the sector allocated to the ith-ranked chromosome. The zeroth-ranked chromosome is the fittest and has the sector with the largest angle A_0, whereas the $(P - 1)$-th-ranked chromosome is the least fit and has the sector with the smallest angle A_{P-1}. The ratio of the sector angles between two adjacently ranked chromosomes is a constant $R = A_i/A_{i+1}$, where $0 \leq i < P - 1$. If the 360 degrees of the wheel are normalized to one, then $A_i = R^{P-i-1} \times (1 - R)/(1 - R^P)$, where $R > 1$, $0 \leq i < P$, and $0 < A_i < 1$.

The selection step generates P random numbers, ranging from zero to one. Each number falls in a sector on the roulette wheel and a copy of the owner chromosome of this sector is included in the next generation. Because a better solution has a larger sector angle than that of a worse solution, there is a higher probability that one or more copies of this better solution will be included in the next generation. In this way, the population for the next generation is determined. Thus, the population size is always P, and it is possible to have multiple copies of the same chromosome.

This GA-based approach also incorporates *elitism* [41]. At the end of each iteration, the best chromosome is always compared with the previous best (elite) chromosome, a copy of which is stored separately from the population. If the best chromosome is better than the elite chromosome, a copy of it becomes the elite chromosome. If the best chromosome is not as good as the elite chromosome, a copy of the elite chromosome replaces the worst chromosome in the population. Elitism is important because it guarantees that the quality of the best solutions found over generations is monotonically nondecreasing.

6.4.5   Crossover Operators

The crossover operator selects a random number of pairs of scheduling strings, where every string has an equal probability of being selected. For each pair, it randomly generates a cutoff point, which divides the scheduling strings of the pair into top and bottom parts. Then, the subtasks in each bottom part are reordered. The new ordering of the subtasks in one bottom part is the relative positions of these
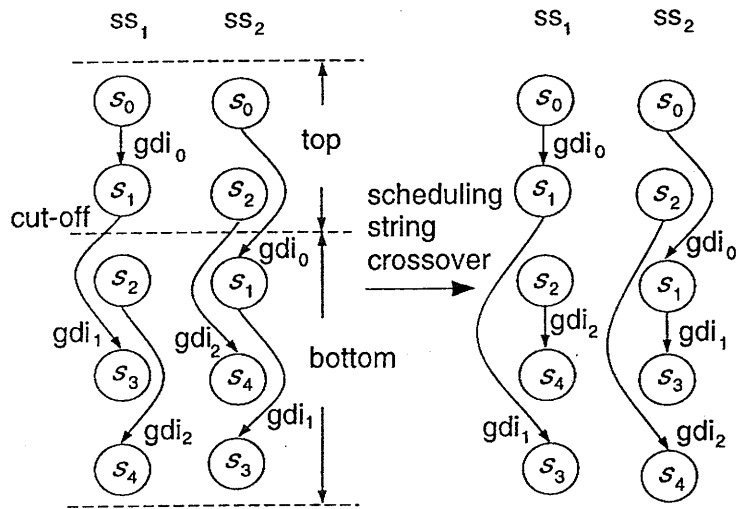
Figure 6.4   A scheduling string crossover example.

subtasks in the other original scheduling string in the pair, thus guaranteeing that the newly generated scheduling strings (which replace the originals) are valid schedules. Figure 6.4 demonstrates such a scheduling string crossover process.

The crossover operator selects a random number of pairs of matching strings, where every string has an equal probability of being selected. For each pair, it randomly generates a cutoff point to divide both matching strings of the pair into two parts. Then the machine assignments of the bottom parts are exchanged, and the new strings replace the originals.

6.4.6   Mutation Operators

The scheduling-string mutation operator selects a random number of scheduling strings, where every string has an equal probability of being selected. Then for each chosen scheduling string, it randomly selects a victim subtask. The *valid range* of the victim subtask is the set of the positions in the scheduling string at which this victim subtask can be placed without violating any data dependency constraints. Specifically, the valid range is after all source subtasks of the victim subtask and before any destination subtask of the victim subtask. After a victim subtask is chosen, it is moved randomly to another position in the scheduling string within its valid range. The new string replaces the original. Figure 6.5 shows an example of this mutation process. For the matching string mutation operator, every matching string has an equal probability of having one of its subtasks assigned to a different, randomly selected, machine.
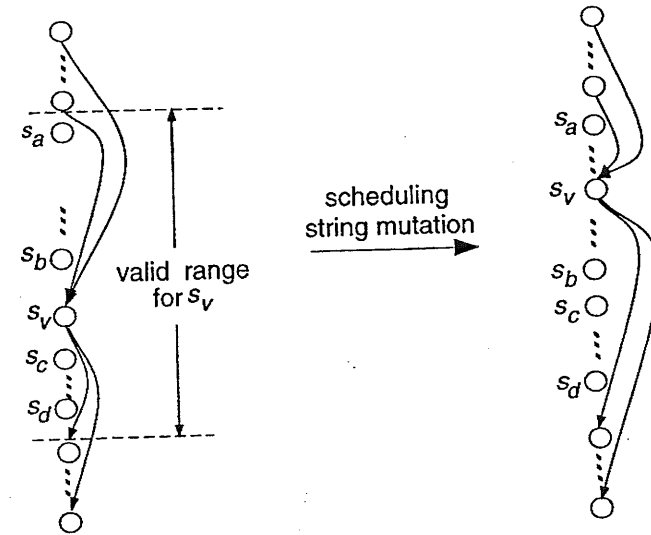
Figure 6.5   A scheduling string mutation example. Only edges to and from the victim subtask s_v are shown. Before the mutation, s_v is between s_b and s_c. After the mutation, it is moved to between s_a and s_b.

6.4.7   Evaluation

The final step of an iteration of a GA is the evaluation of each chromosome to determine its fitness value. In this GA-based approach, the chromosome structure is independent of any particular communication subsystem. Only the evaluation step needs the communication characteristics of the given HC system to schedule the data transfers.

To test the effectiveness of this GA-based approach, an example communication system modeled after a HiPPI LAN with a central crossbar switch [28, 53] is assumed to connect a suite of machines. Each machine in the HC suite has one input data link and one output data link. All these links are connected to a central crossbar switch. If a subtask needs a gdi that is produced or consumed earlier by a different subtask on the same machine, the communication time for this item is zero. Otherwise, the communication time is obtained by dividing the size of the gdi by the smaller bandwidth of the output link of the source machine and the input link of the destination machine. In this research, it is assumed that for a given machine the bandwidths of the input link and the output link are equal to each other. It is also assumed that the crossbar switch has a higher bandwidth than that of each link. The communication latency between any pair of machines is assumed to be the same. Data transfers are neither preemptive nor multiplexed. Once a data-transfer path is established, it cannot he relinquished until the data item (i.e., some gdi_k) scheduled to be transferred over this path is received by the destination machine. Multiple data transfers over the same path must be serialized.

In the evaluation step, for each chromosome the final order of execution of the subtasks and the intermachine data transfers are determined. The evaluation procedure considers the subtasks in the order they appear on the scheduling string. Subtasks assigned to the same machine are executed exactly in the order specified by the scheduling string. For subtasks assigned to different machines, the actual execution order may deviate from that specified by the scheduling string due to factors such as input-data availability and machine availability. This is explained below.

Before a subtask can be scheduled, all of its input gdis must be received. For each subtask, its input data items are considered by the evaluation procedure in the order of their producers' relative positions in the scheduling string. In Fig. 6.6, a simple example is shown to illustrate the evaluation for a given chromosome. In this example (as well as some others given later), because there are only two machines, the source and destination machines for the gdi transfers are implicit.

When a subtask to be scheduled has multiple input gdis that have not been received, the gdi whose producer subtask is listed earliest in the scheduling string is considered first. The reason for this ordering is to attempt to better utilize the overlap of subtask executions and intermachine data communications. The following example illustrates this idea. Let $ss(0) = s_0$, $ss(1) = s_1$, $ss(2) = s_2$, as shown in Fig. 6.7a. Let s_2 need two gdis, gdi_0 and gdi_1, from s_0 and s_1, respectively. Depending on the subtask to machine assignments, the data transfers of gdi_0, and gdi_1 could be either local, i.e., within a machine, or across machines. If at least one data transfer is local, then the scheduling is trivial because it is assumed that local transfers within a machine take negligible time. However, there exist two situations where both data transfers are across machines so that they need to be ordered.

Situation 1   Let s_0 and s_1 be assigned to the same machine m_0 and s_2 be assigned to another machine m_1, as shown in Fig. 6.7b. In this situation, because s_0 is to be executed before s_1, gdi_0 is available before gdi_1 becomes available on machine m_0. Thus, it is better to consider the gdi_0 transfer before the gdi_1 transfer.

Situation 2   Let the three subtasks s_0, s_1, and s_2 be assigned to three different machines m_0, m_1, and m_2, as shown in Fig. 6.7c. In this situation, if there is a data dependency from s_0 to s_1, then s_0 finishes its execution before s_1 could start. Therefore, gdi_0 is available before gdi_1 becomes available. Hence, it is better to consider the gdi_0 transfer before the gdi_1 transfer. If there are no data dependencies from s_0 to s_1, the gdi_0 transfer can still be considered before the gdi_1 transfer. While this may not be necessary in this case, it is reasonable to do because there may be some other chromosome(s) that have $ss(0) = s_1$ and $ss(1) = s_0$. When such a chromosome is evaluated, the gdi_1 transfer will be considered before the gdi_0 transfer. Therefore, it is possible for all input gdi scheduling orderings for gdi_0 and gdi_1 to be examined.

Data forwarding is another important feature of this evaluation process. For each input data item to be considered, the evaluation process chooses the source subtask
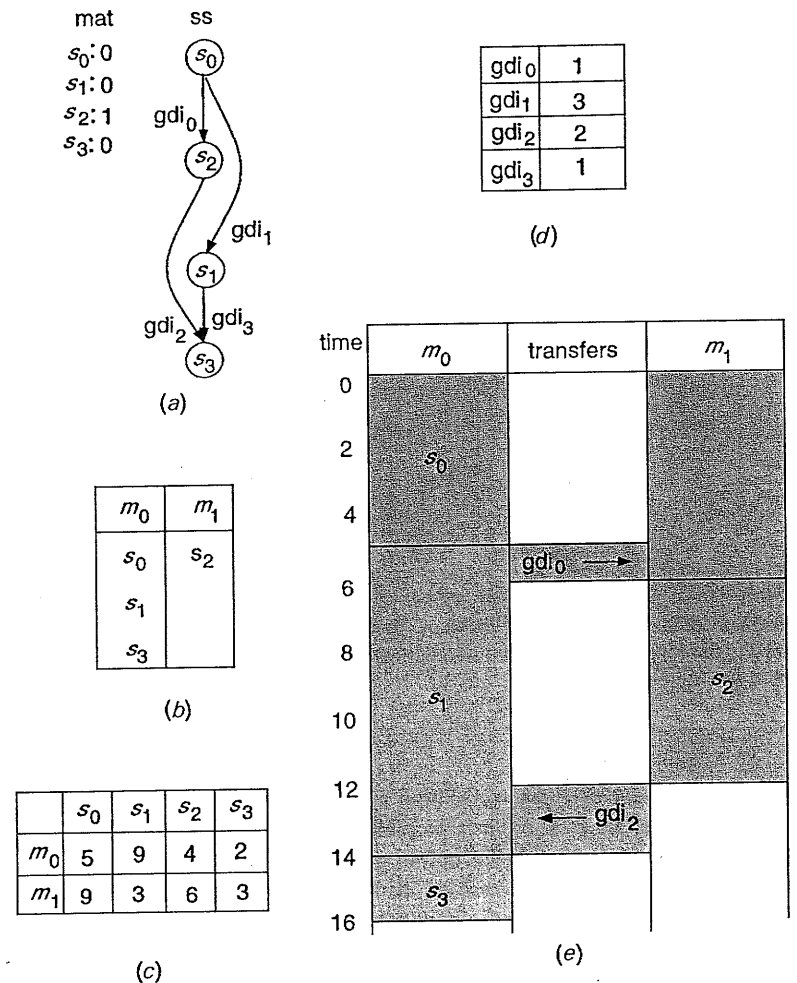
Figure 6.6   A very simple example showing the evaluation step: (*a*) the chromosome; (*b*) the subtask execution ordering on each machine given by (*a*); (*c*) the estimated subtask execution times; (*d*) the gdi intermachine transfer times (transfers between subtasks assigned to the same machine take zero time); and (*e*) the subtask execution and data-transfer timings, where the completion time for this chromosome is 16.

from among the producer of this data item and all the consumers that have received this data item. These consumers are *forwarders* [51]. The one (either the producer or a forwarder) from which the destination subtask will receive the data item at the earliest possible time is chosen.

After the source subtask is chosen, the data transfer for the input data item is scheduled. A transfer starts at the earliest point in time when the path from the
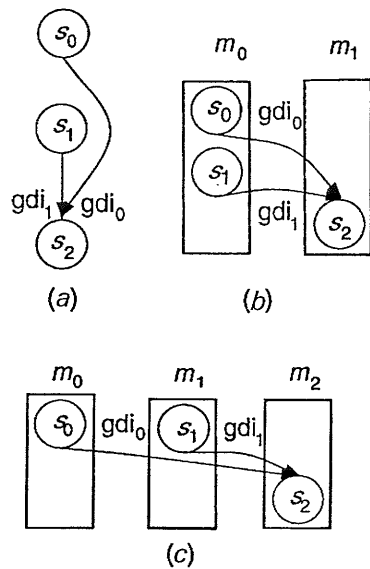
**Figure 6.7** An example scheduling string with two possible matching strings: (a) the example scheduling string; (b) the situation when the source subtasks of the input gdis are assigned to the same machine; (c) the situation when the source subtasks of the input gdis are assigned to different machines.

source machine to the destination machine is free for a period that is at least equal to the needed transfer time. Thus, it is possible that, for example, gdi_1 is considered before gdi_2, but gdi_2 is transferred before gdi_1. This is referred to as *out-of-order* scheduling of data transfers because the data transfers do not occur in the order in which they are considered (i.e., the *in-order* schedule). This (possibly) out-of-order scheduling of the input item data transfers utilizes previously idle bandwidths of the communication links, and thus could make some input data items available to some subtasks earlier than the in-order scheduling. As a result, some subtasks could start their execution earlier, which would in turn decrease the overall task completion time. Figures 6.8 and 6.9 show the in-order scheduling and the out-of-order scheduling for the same chromosome, respectively. In the in-order scheduling, the transfer of gdi_1 is scheduled before the transfer of gdi_2 because subtask s_2 input data transfers are considered before those of subtask s_3. In this example, the out-of-order schedule does decrease the total execution time of the given task.

When two chromosomes have different matching strings, they are different solutions because the subtask-to-machine assignments are different. However, two chromosomes that have the same matching string but different scheduling strings may or may not represent the same solution. This is because the scheduling-string information is used in two cases: (1) for scheduling subtasks that have been assigned
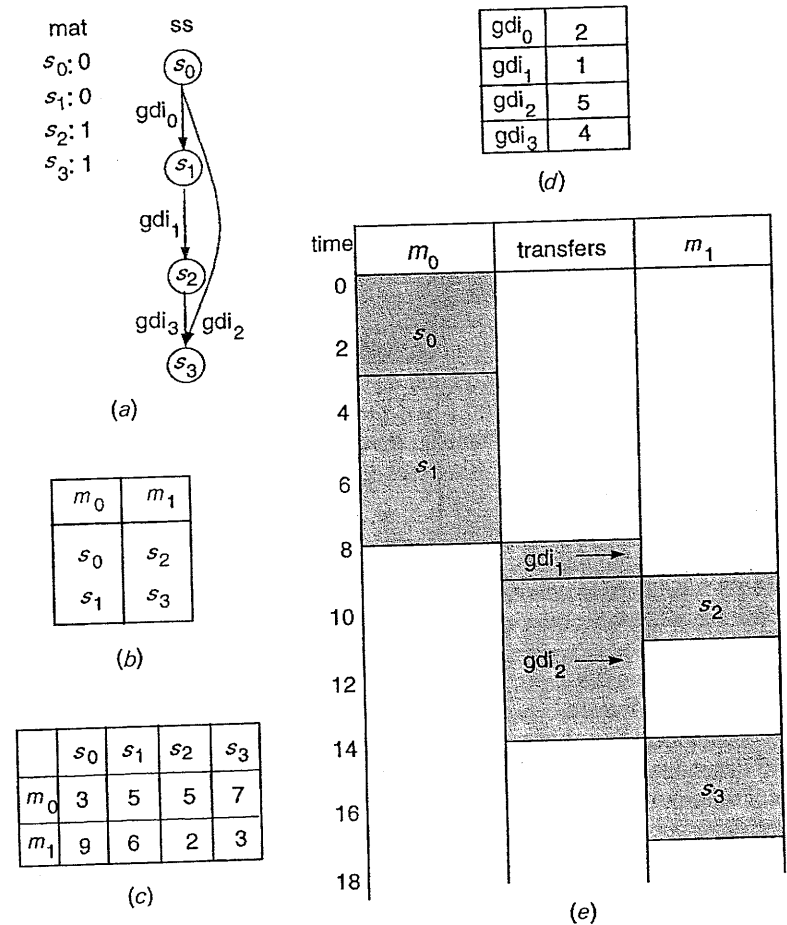
Figure 6.8 An example showing the in-order scheduling of a chromosome: (a) the chromosome; (b) the subtask execution ordering on each machine; (c) the estimated subtask execution times; (d) the gdi transfer times (transfers between subtasks assigned to the same machine take zero time); and (e) the subtask execution and data-transfer timings using in-order transfers (the gdi_1 transfer occurs before the gdi_2 transfer), where the completion time is 17.

to the same machine, and (2) for examining data transfers. Two different scheduling strings could result in the same ordering for (1) and (2).

After a chromosome is evaluated, it is associated with a fitness value, which is a time when the last subtask finishes its execution. That is, the fitness value of a chromosome is the overall execution time of the task, given the mapping decision specified by this chromosome and by the evaluation process.

In summary, this evaluation mechanism considers subtasks in the order in which they appear in the scheduling string. For a subtask that requires some gdis from other
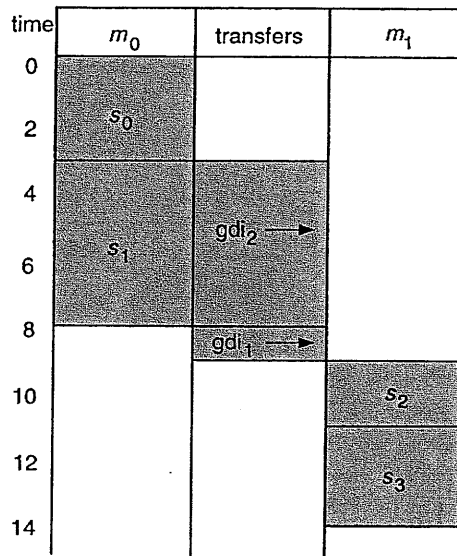
Figure 6.9 An example showing the the out-of order scheduling, where the chromosome and other statistics are the same as in Fig. 6.8. The completion time is 14.

machines, the gdi transfer whose producer subtask appears earliest in the scheduling string is examined first. When scheduling a gdi transfer, both the producing and the forwarding subtasks are considered as possible sources. The source subtask that lets this consumer subtask receive this gdi at the earliest possible time is chosen to send the gdi. The out-of-order scheduling of the gdi transfers over a path could further reduce the completion time of the application.

6.4.8 Experimental Results

To measure the performance of this GA-based approach, randomly generated scenarios were used, where each *scenario* corresponded to an SPDAG, the associated subtask execution times, the sizes of the associated gdis, and the communication-link bandwidths of the machines. The scenarios were generated for different numbers of subtasks and different numbers of machines, as specified below. The estimated expected execution time for each subtask on each machine, the number of gdis, the size of each gdi, and the bandwidth of each input link of each machine were randomly generated with uniform probability over some predefined ranges. For each machine, the bandwidth of the output link is made equal to that of the input link. The producer and consumers of each gdi were also generated randomly. The scenario generation used a $|G| \times |S|$ dependency matrix to guarantee that the precedence constraints from data dependencies were acyclic.

These randomly generated scenarios were used for three reasons: (1) it is desirable to obtain data that demonstrate the effectiveness of the approach over a broad range of conditions, (2) a generally accepted set of HC benchmark tasks does not exist, and (3) it is not clear what characteristics a "typical" HC task would exhibit [56]. Determining a representative set of HC task benchmarks remains a challenge for the scientific community in this research area.

In this work, small-scale and larger scenarios were used to quantify the performance of this GA-based approach. The scenarios were grouped into three categories, namely tasks with light, moderate, and heavy communication loads. A lightly communicating task has its number of gdis in the range of $0 \le |G| < (1/3)|S|$; a moderately communicating task has its number of gdis in the range of $(1/3)|S| \le |G| < (2/3)|S|$; and a heavily communicating task has its number of gdis in the range of $(2/3)|S| \le |G| < |S|$. The ranges of the gdi sizes and the estimated subtask execution times were both from 1 to 1,000. For these scenarios, the bandwidths of the input and output links were randomly generated, ranging from 0.5 to 1.5. Hence, the communication times in these scenarios were source and destination machine dependent.

The probability of crossover was the same for the matching string and the scheduling string. The probability of mutation was also the same for the matching string and the scheduling string. The stopping criteria were (1) the number of iterations had reached 1,000, (2) the population had converged (i.e., all the chromosomes had the same fitness value), or (3) the currently best solution had not improved over the last 150 iterations. All the GA runs discussed in this section stopped due to their best solutions not improving for 150 iterations.

The GA-based approach was first applied to 20 small-scale scenarios that involved up to ten subtasks, three machines, and seven global data items. The GA runs for small-scale scenarios had the following parameters. The probabilities for scheduling-string and matching-string crossovers, were both chosen to be 0.4, and scheduling-string and matching-string mutations were both chosen to be 0.1. The GA population size, P, for small-scale scenarios was chosen to be 50. The angle ratio of the sectors on the roulette wheel for two adjacently ranked chromosomes, R, was chosen to be $1 + 1/P$. By using this simple formula, the angle ratio between the slots of the best and median chromosomes for $P = 50$ (and also for $P = 200$ for larger scenarios discussed later in this section) was very close to the optimal empirical ratio value of 1.5 in [60].

The results from a small-scale scenario are used here to illustrate the search process. This scenario had $|S| = 7$, $|M| = 3$, and $|G| = 6$. The SPDAG, the estimated execution times, and the transfer times of the global data items are shown in Fig. 6.10. The total numbers of possible different matching strings and different valid scheduling strings (i.e., topological sorts of the SPDAG) were $3^7 = 2187$ and 16, respectively. Thus, the total search space had $2187 \times 16 = 34,992$ possible chromosomes.

Exhaustive searches were performed to find the optimal solutions for the small-scale scenarios. For each of the small-scale scenarios that were conducted, the GA-based approach found one or more optimal solutions that had the same completion
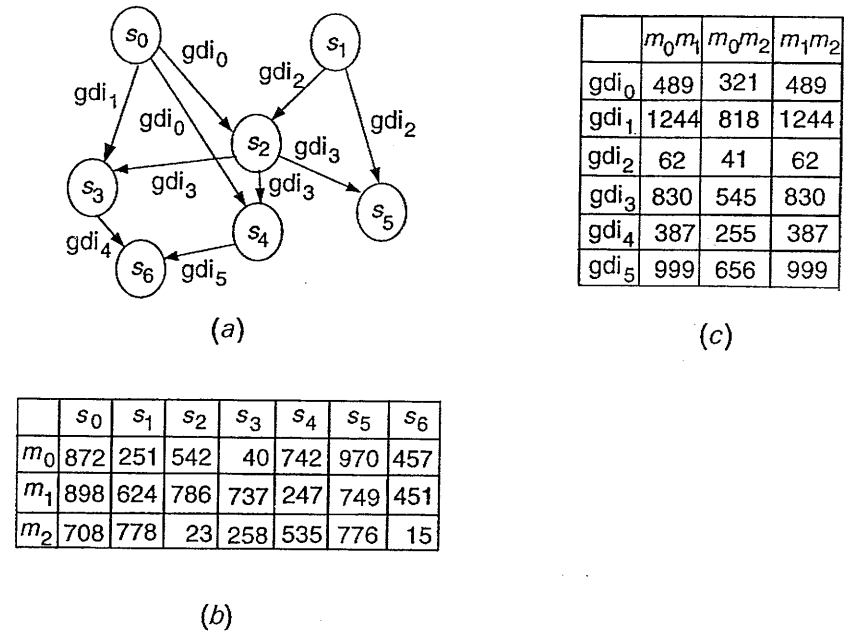
(a)

	$m_0 m_1$	$m_0 m_2$	$m_1 m_2$
gdi_0	489	321	489
gdi_1	1244	818	1244
gdi_2	62	41	62
gdi_3	830	545	830
gdi_4	387	255	387
gdi_5	999	656	999

(c)

	s_0	s_1	s_2	s_3	s_4	s_5	s_6
m_0	872	251	542	40	742	970	457
m_1	898	624	786	737	247	749	451
m_2	708	778	23	258	535	776	15

(b)

Figure 6.10 A small-scale simulation scenario: (a) the SPDAG; (b) the estimated execution times; and (c) the transfer times of the gdis from a given source machine to a given destination machine.

time, verified by the best solution(s) found by the exhaustive search. The GA search for a small-scale scenario that had ten subtasks, three machines, and seven gdis took about one minute to find multiple optimal solutions on a Sun Sparc5 workstation, while the exhaustive search took about eight hours to find these optimal solutions.

The performance of this GA-based approach was also examined using larger scenarios with up to 100 subtasks and 20 machines. These larger scenarios were generated using the same procedure as for generating the small scenarios except that the GA population size for larger scenarios was chosen to be 200.

Larger scenarios are intractable problems. It is currently impractical to directly compare the quality of the solutions found by the GA-based approach for these larger scenarios with those found by exhaustive searches. It is also often difficult to compare the performance of different HC task mapping approaches due to the different HC system models that are assumed by researchers when they design mapping heuristics. However, the model used in [30] is similar to the one being used in this research work. Hence, the performance of the GA-based approach on larger scenarios was compared with the nonevolutionary *levelized min-time* (*LMT*) heuristic proposed in [30]. (In Section 6.6, a variety of heuristics are compared by adapting them for a simple common model.)

The LMT heuristic first establishes levels of subtasks in the following way. The subtasks that have no input gdis are at the highest level. Each of the remaining subtasks is at one level below the lowest producer of its gdis. The subtasks at the highest level are to be considered first. The LMT heuristic averages the estimated execution times for each subtask across all machines. At each level, a level-average execution time i.e., the average of the machine-average execution times of all subtasks at this level, is also computed. If there are some levels between a subtask and its closest child subtask, the level-average execution time of each middle level is subtracted from the machine-average execution time of this subtask. The adjusted machine-average execution times of the subtasks are used to determine the priorities of the subtasks within each level, i.e., a subtask with a larger average is to be considered earlier at its level. If the number of subtasks at a level is greater than the number of machines in the HC suite, the subtasks with smaller averages are merged so that as a result, the number of combined subtasks at each level equals the number of machines available. When a subtask is being considered, it is assigned to the fastest machine available from those machines that have not yet been assigned any subtasks from the same level.

Another nonevolutionary heuristic, the *baseline* (*BL*), was developed as part of this GA research, and the solution it found was incorporated into the initial population. Similar to the LMT heuristic, the BL heuristic first establishes levels of subtasks based upon their data dependencies. Then all subtasks are ordered such that a subtask at a higher level comes before one at a lower level. The subtasks in the same level are arranged in descending order of their number of output gdis (ties are broken arbitrarily). The subtasks are then scheduled as follows. Let the ith subtask in this order be σ_i, where $0 \leq i < |S|$. First, subtask σ_0 is assigned to a machine that gives the shortest completion time for σ_0. Then, the heuristic evaluates $|M|$ assignments for σ_1, each time assigning σ_1 to a different machine, with the previously decided machine assignment of σ_0 left unchanged. The subtask σ_1 is finally assigned to a machine that gives the shortest overall completion time for both σ_0 and σ_1. The BL heuristic continues to evaluate the remaining subtasks in the order defined earlier. When scheduling subtask σ_i, $|M|$ possible machine assignments are evaluated, each time with the previously decided machine assignments of subtasks σ_j $(0 \leq j < i)$ left unchanged. Subtask σ_i is finally assigned to a machine that gives the shortest overall completion time of subtasks σ_0 through σ_i. The total number of evaluations is thus $|S| \times |M|$, and only i subtasks (out of $|S|$) are considered when performing evaluations for the $|M|$ machine assignments for subtasks σ_i.

To determine the best GA parameters for solving larger HC mapping problems, 50 larger scenarios were randomly generated in each communication category. Each of these scenarios contained 50 subtasks and 5 machines. For each scenario, GA runs were conducted for the following combinations of crossover probability and mutation probability. The crossover probability ranged from 0.1 to 1.0 in steps of 0.1, and the mutation probability ranged from 0.04 to 0.40 in steps of 0.04 and from 0.4 to 1.0 in steps of 0.1. Let the *relative solution quality* be the task completion time of the solution found by the LMT heuristic divided by that found by the approach being investigated. A greater value of the relative solution quality means that the

approach being investigated finds a better solution to the HC mapping problem (i.e., with a shorter overall completion time for the application task represented by the SPDAG). With each crossover and mutation probability pair and for each communication load, the average relative solution quality of the 50 GA runs, each on a different scenario, was computed.

For each communication load category, a *region of good performance* could be identified for a range of crossover and mutation probabilities. The regions of good performance generally consisted of moderate-to-high crossover probability and low-to-moderate mutation probability. This is consistent with the results from the GA literature, which show that crossover is GA's major operator, and mutation plays a secondary role in GA searches [14, 25, 49].

The crossover and mutation probabilities with which the best relative solution quality had been achieved were used in each corresponding communication load category. When mapping real tasks, the communication load can be determined by computing the ratio of the number of gdis to the number of subtasks. Once the communication load category is known, a probability pair from the corresponding region of good performance can be used.

On Sun Sparc5 workstations, for these larger scenarios, both the LMT heuristic and the BL heuristic took no more than one minute of CPU time to execute. The average CPU execution time of the GA-based approach on these scenarios ranged from less than one minute for the smallest scenarios (i.e., five subtasks, two machines, and light communication load) to about three and one-half hours for the largest scenarios (i.e, 100 subtasks, 20 machines, and heavy communication load). Recall that it is assumed that this GA-based approach will be used for application tasks that are large production jobs such that the one-time investment of this high execution time is justified.

The performance of the GA-based approach was also compared with that of a random search. For each iteration of the random search, a chromosome was randomly generated, this chromosome was evaluated, and the fitness value was compared with the saved best fitness value. If the fitness value of the current chromosome was better than the saved best value, it became the saved best fitness value. For each scenario, the random search iterated for the same length of time as that taken by the GA-based approach on the same scenario.

Figure 6.11 shows the performance comparisons between the LMT heuristic and the GA-based approach for heavily communicating larger scenarios. The GA-based approach used a crossover probability of 1.0 and a mutation probability of 0.2. These values were the best performing probabilities over a wide range of combinations examined for a separate set of test scenarios. In the figure, the horizontal axes are the number of subtasks in log scale. The vertical axes are the relative solution quality of the various approaches. Each point in the figure is the average of 50 independent scenarios. The performance comparisons among the GA-based approach, the LMT heuristic, the BL heuristic, and the random search for moderately communicating and lightly communicating larger scenarios can be found in [58].

In all cases, the GA-based approach presented here outperformed the BL and LMT heuristics and the random search. The improvement of the GA-based approach
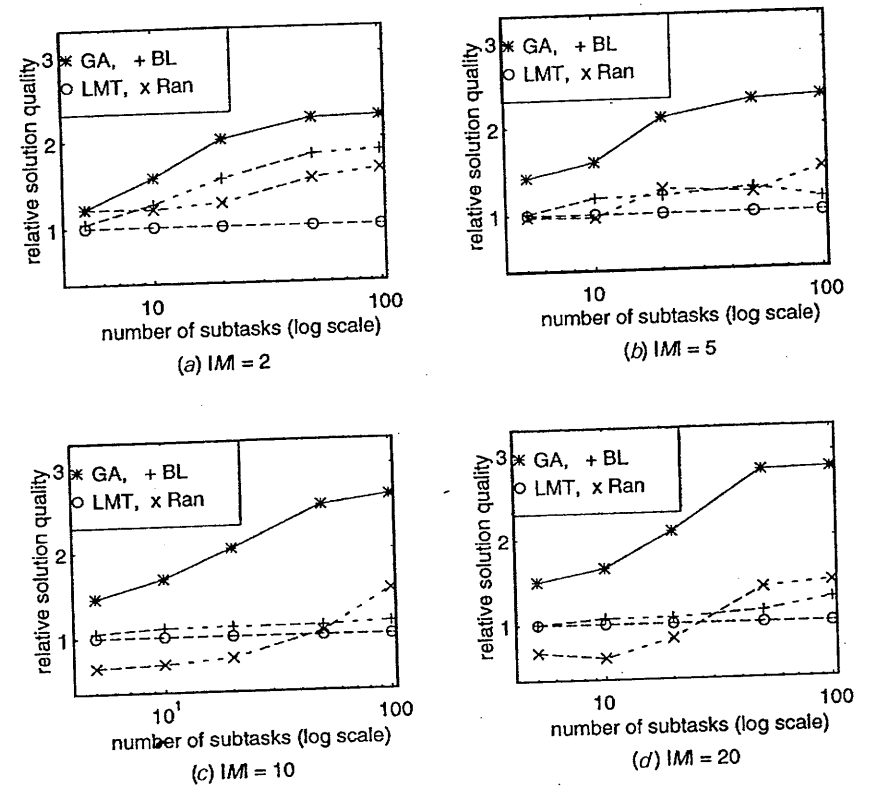
Figure 6.11 Performance comparisons of the GA-based approach with a crossover probability of 1.0 and a mutation probability of 0.2, relative to the LMT heuristic for heavily communicating larger scenarios in (*a*) a two-machine suite, (*b*) a five-machine suite, (*c*) a ten-machine suite, and (*d*) a 20-machine suite. The relative performance of the BL heuristic and the random search are also shown.

over the others showed an overall trend to increase as the number of subtasks increased. The exact shape of the GA-based-approach performance curves is not as significant as the overall trends because the curves are for a heuristic operating on randomly generated data, resulting in some varied performance even when averaged over 50 scenarios for each data point.

6.4.9 Summary

A novel GA-based approach for task mapping in HC environments was presented. This GA-based approach can be used in a variety of HC environments because it does not rely on any specific communication subsystem model. It is applicable to the static scheduling of production jobs, and can be readily used for scheduling multiple

independent tasks (and their subtasks) collectively. For small-scale scenarios, the proposed approach found optimal solutions. For larger scenarios, it outperformed two nonevolutionary heuristics and a random search.

There are a number of ways this GA-based approach for HC task mapping can be built upon for future research. These include extending this approach to allow multiple producers for each of the global data items, parallelizing the GA-based approach, developing evaluation procedures for other communication subsystems, and considering loop and data-conditional constructs that involve multiple subtasks.

6.5 SEMISTATIC MATCHING AND SCHEDULING OF SUBTASKS

6.5.1 Overview

In this section, the application domain and HC platform assumed for the semistatic approach are described [7, 8]. This is followed by a discussion of the enhancements made to the GA described in Section 6.4 for determining off-line mappings in this situation. Finally, Section 6.5.7 summarizes the results of an extensive performance study of a simulated semistatic mapping system [35].

6.5.2 Application Domain: Automatic Target Recognition

One type of automatic target recognition (ATR) system takes a stream of images from a group of sensors and produces a scene description [54], tracking the movement of possible targets through the sequence of images. A simplified example of an ATR task can be found in [35]. The various types of image-processing elements required in an ATR system can be broadly classified into three groups: low-level processing (numeric computation), intermediate-level processing (quasi-symbolic computation, e.g., where numeric and symbolic types of operations are used to describe surfaces and shapes of objects in the scene), and high-level processing (symbolic computation, e.g., used to produce the scene description) [1, 4]. Each of these subtasks may allow the use of multiple processors of the same type for efficient execution. Heterogeneous parallel architectures consisting of multiple copies of different types of processors (e.g., SHARC DSP and PowerPC RISC processors [16]) are appropriate computing platforms for efficiently handling computational tasks with such diverse requirements.

The class of ATR applications considered can be modeled as an iterative execution (one execution for each image in the stream) of a set of partially ordered subtasks. Thus, the ATR task can be modeled by a DAG in which the nodes represent subtasks and the edges represent the communications among subtasks. It is assumed that an ATR application in this class is a production job that is executed repeatedly. Therefore, it is worthwhile to invest extra off-line time in preparing an effective mapping of the application onto the hardware platform used to execute it.

6.5.3 An Intelligent Operating System

A major distinctive architectural feature of the envisioned IOS is the capability for the on-line use of off-line computed mappings [7, 8, 35]. The *ATR Kernel* makes decisions on how a given ATR application task should be accomplished, including determining the partial ordering of subtasks and which algorithms could be used to accomplish each subtask. The *HC Kernel* uses a semistatic method to decide how the partially ordered algorithmic suggestions should be implemented and mapped onto the heterogeneous parallel platform. Furthermore, the HC Kernel interacts with the *Basic Kernel* (the low-level operating system) to initiate the application and monitor its execution. This allows the HC Kernel to decide at the end of each iteration through the application if the subtasks should be remapped onto the hardware platform. Information from the *Algorithm Database* and the *Knowledge Base* is used to support the ATR and HC Kernels.

The Algorithm Database includes one or more implementations of each algorithm (e.g., one for each processor type). The Algorithm Database also contains the expected execution time of each algorithm implementation, typically specified as a function of type and number of processors assigned, interprocessor communication time, and certain input data and application characteristics. Dynamic parameters are those input characteristics, such as amount of clutter, and application characteristics, such as number of located objects to be identified, that (1) will change during run time, (2) can be computed by the application as it executes, and (3) can impact the execution time of each subtask in the task graph. This is an expected time rather than a definite time, because it can vary depending on the actual values of the input data being processed. System parameters, such as the number of each type of processor in the platform, are stored in the Knowledge Base. The off-line and on-line components of the HC Kernel are detailed below. For more details about the IOS, the reader is referred to [8].

6.5.4 Off-line HC Kernel Component: MDDG Generator

A *mapped data-dependency graph* (*MDDG*) is a DAG annotated with the task-to-machine mapping. The *HC Kernel MDDG Generator*, which is a major component of the off-line IOS, is responsible for mapping each DAG onto the heterogeneous parallel hardware platform creating a corresponding MDDG. The structure of the HC Kernel MDDG Generator is such that any effective heuristic could be employed. For the iterative ATR application domain, it is possible to use off-line precomputed mappings to reconfigure resources in real time.

Because different sets of dynamic parameter values can lead to different subtask execution times (e.g., more objects to be recognized in the scene), the HC Kernel MDDG Generator will, in general, generate different mappings for the same DAG (corresponding to different sets of dynamic parameter values). It is assumed that the ranges of the dynamic parameters are known. The space of dynamic parameters is partitioned into a number of disjoint regions, and within each region a random set of representative dynamic parameter vectors are chosen, each of which is called a

sample vector. For each sample vector, an off-line mapping heuristic (the enhanced GA described in Section 6.5.6) uses the following information to create a mapping, i.e., to transform a DAG into an MDDG: (1) the structure of the underlying DAG; (2) the expected execution time of each subtask on a set of processors (of the same type) assigned, as a function of the type and number of processors; (3) the intersubtask data transfers needed, in terms of formats and expected sizes of the data items to be transferred; (4) the expected time to send data from one processor to another as a function of the size of the data item to be transferred; and (5) the number of each type of processor that is in the hardware platform.

The mapping for each sample vector is exhaustively evaluated for every other sample vector in the region by applying the mapping to the DAG and computing the task execution time using that other sample vector's dynamic parameter values. The mapping that gives the minimum average execution time is chosen as the representative mapping for the corresponding region in the dynamic parameter space. This representative mapping and the corresponding average execution time are stored in the off-line mapping table (i.e., the *MDDG Table*), which is a multidimensional array, indexed by dynamic parameter ranges. Thus, for a given DAG, the MDDG for each region of the dynamic parameter space is stored in the MDDG table for that DAG, along with the corresponding expected execution time.

6.5.5   On-Line HC Kernel Component: Execution Monitor

The *HC Kernel Monitor* is an on-line component responsible for (1) establishing the initial mapping of the given application onto the hardware platform, and (2) monitoring the execution of the application and, at the end of each iteration through the corresponding DAG, deciding if and how the mapping of the application onto the hardware platform should be changed based on information about the actual values of the dynamic parameters. Examples of dynamic parameters include the contrast level of an image, the number of objects in a scene, and the average size (in pixels) of an object in a scene.

The initial mapping is selected from the MDDG Table for that DAG based on menu-driven input from the application user about an initial value to assume for each of the dynamic parameters. During execution of the application, the HC Kernel Monitor receives actual updated values of the dynamic parameters at the end of each iteration through the corresponding DAG. The HC Kernel Monitor will use the most recent values of these dynamic parameters to estimate if changing the mapping will reduce the expected execution time of the next iteration through the corresponding DAG. It does this by comparing the actual execution time of the last completed iteration with the sum of an estimated time for reconfiguration plus the expected execution time from the MDDG Table entry for the region that includes the most recent dynamic parameter values known. Thus, this decision is made in real time after all subtasks' implementations for the current iteration have finished executing and before any subtask implementations begin to execute for the next iteration. If it is desirable to change the mapping, then the HC Kernel Monitor will pass the

MDDG Table entry index to the Basic Kernel for use in the next iteration. If not, the same mapping will continue to be used.

6.5.6   Genetic Algorithm Used for Semistatic Approach

The genetic algorithm described in Section 6.4 is enhanced for determining the off-line mappings. Because each subtask can be mapped to multiple processors of the same type, a new string, called the *allocation string*, is also incorporated into each chromosome in the enhanced GA. Specifically, each entry i in the allocation string represents the number of processors of a certain type (specified by the corresponding entry in the matching string) assigned to the subtask s_i. Typically, multiple subtasks will be assigned to some of the same processors in a processor group. The subtasks are then executed in a nonpreemptive manner based on the ordering that is specified by the scheduling string.

The initial allocation string for each chromosome is generated by randomly selecting a value from 1 to p_{opt} (defined in Section 6.5.7.2) as the number of processors allocated to each subtask (for the processor group type specified in the matching string). The solution generated by a fast heuristic is also included in the initial population. The heuristic used is a fast static scheduling algorithm, called the *earliest completion time (ECT)* algorithm [57], which is similar to the LMT heuristic discussed in Section 6.4.8 and is described in [35]. One chromosome with an allocation string where each entry is the optimal number of processors for that subtask (individually) is also included in the initial population.

The mutation operator for the allocation string randomly selects an entry in the string and locally optimizes it by changing the number of processors to a value that gives the best total task execution time. The crossover operator for allocation strings is the same as the one used for matching strings. The probability for mutation and crossover for all strings in the experiments below is 0.4.

The GA is executed 10 times for each sample vector. To enhance diversity, only 5 of the 10 runs include a chromosome generated by the ECT algorithm in the initial population.

6.5.7   Performance Results

6.5.7.1 Overview   In [34, 35], an extensive performance study was performed on a simulated HC Kernel implemented using the enhanced GA as the off-line heuristic. The goal of the study was to evaluate the ideas underlying the particular semistatic method of on-line use of off-line-derived mappings described earlier (referred to an *On–Off* in subsequent sections). Four approaches were compared in the experiments: (1) the On–Off approach; (2) the ECT algorithm as a dynamic scheduling algorithm; (3) the infeasible approach of using the GA as a dynamic scheduling algorithm (referred to as *GA On-line*) and (4) an ideal but impossible approach that uses the GA On-line with the exact (as yet unknown) dynamic parameters for the iteration to be executed next (referred to as *Ideal*). In the latter two schemes, a mapping determined for a previous iteration is also included in the initial

population of the current iteration, and reconfiguration times are ignored. These two infeasible schemes were merely used as references for comparison to solution quality of the former two approaches.

6.5.7.2 System Model

To evaluate the On–Off semistatic mapping methodology, an example architecture was chosen [35]. It consisted of 4 different types of processors with 16 processors for each type (i.e., total number of processors in the HC platform is 64). The processors within each type are connected via a 17-port crossbar switch, whereas the 4 different types of processor are connected using a 4-port crossbar switch. The execution time of a subtask and the communication time between subtasks in an application task were modeled by equations that are functions of the dynamic parameters. Specifically, the simple execution time expression used in this task model is a version of Amdahl's law extended by a term representing the parallelization overhead (e.g., synchronization and communication). The serial and parallel fractions of a subtask are frequently represented using similar models (e.g., [9, 43]). The execution-time expression for subtask s_i includes: (1) three generic dynamic parameters, α, β, and γ; (2) the number of processors used, p; and (3) three coefficients, a_i, b_i, and c_i. The parallel fraction and serial fraction of subtask s_i are represented by $a_i\alpha/p$ and $c_i\gamma$, respectively. The parallelization overhead is represented by $b_i\beta \log p$. The relative speed of a subtask s_i on a processor of type u is given by the heterogeneity factor h_{iu}. The execution time of subtask s_i is then given by the expression: $b_{iu}(a_i\alpha/p + b_i\beta \log p + c_i\gamma)$. By differentiating this equation and equating it to zero, the optimal value of p (p_{opt}) that leads to the minimum execution time for a given subtask is $(a_i\alpha)/(b_i\beta)$.

It is assumed that the size of the data to be transferred between two subtasks s_i and s_j consists of a fixed portion d_{ij} and a variable portion $e_{ij}\mu$, where μ is a fourth dynamic parameter. For communication between virtual machines whose processor types are u and v, s_{uv} and R_{uv} are the message start-up time and the data transmission rate, respectively. The intersubtask communication time c_{uv} between subtask s_i on a virtual machine with processors of type u and subtask s_j on a virtual machine with processors of type v is given by the expression: $S_{uv} + (d_{ij} + e_{ij}\mu)/R_{uv}$.

The example architecture and the simplified generic equations are used as input to the simulated HC Kernel only. The On–Off method can be adopted for other target architectures and the actual time equations specified by the application developer. In practice, a particular single dynamic parameter can impact any subset of the components of a given subtask's execution time equation.

6.5.7.3 Workload

To investigate the performance of the On–Off approach, randomly generated DAGs containing 10, 50, 100, or 200 were used. These DAGs included regularly structured graphs (in-trees, out-trees, and fork-joins [34, 35]) and randomly structured DAGs. For each size and structure, 10 different graphs were generated, and thus a total of 160 graphs were used. The input to the simulated on-line module consists of an execution profile that is composed of a certain number of iterations of executing the DAG. Examples of two randomly generated execution profiles containing 20 iterations are shown in Table 6.1. In each profile, the dynamic

TABLE 6.1 Execution Profiles of Dynamic Parameters

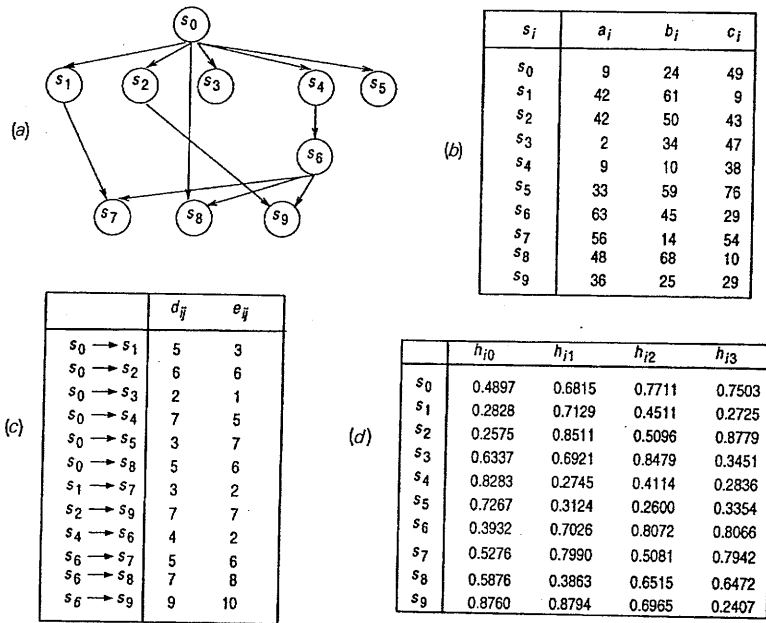|	Profile A					Profile B			
Iteration	α	β	γ	μ	Iteration	α	β	γ	μ
0	3,000	15	300	60	0	3,000	15	300	60
1	2,821	15	287	63	1	4,309	15	409	82
2	2,949	12	302	65	2	2,635	7	268	43
3	3,073	12	286	68	3	3,894	6	361	27
4	3,228	11	273	71	4	2,241	8	197	39
5	3,090	13	258	67	5	1,265	12	287	52
6	3,256	11	272	70	6	1,699	16	420	75
7	3,424	16	259	73	7	1,138	11	282	50
8	3,621	16	271	75	8	1,543	12	153	67
9	3,811	13	260	78	9	2,205	17	225	97
10	4,014	17	245	81	10	3,198	10	332	51
11	4,229	13	257	77	11	4,678	18	477	73
12	3,994	19	242	80	12	2,588	8	315	48
13	4,179	15	253	83	13	1,358	16	211	67
14	4,386	15	264	78	14	1,794	17	307	98
15	4,208	13	249	82	15	2,605	11	163	61
16	4,016	14	236	77	16	3,719	17	240	87
17	3,835	16	226	81	17	2,478	9	332	53
18	4,026	19	238	84	18	1,507	16	466	76
19	4,258	16	251	88	19	2,081	8	243	50
20	4,479	15	265	92	20	3,053	17	149	70

Notes: Profile A, average percentage change in dynamic parameter values = 5%. Profile B, average percentage change = 40%.

parameter values change from one iteration to another within certain ranges (α: [1,000–5,000], β: [5–25], γ: [100–500], and μ: [20–100]). Specifically, the average percentage change in dynamic parameter values in Profile A and Profile B are 5% and 40%, respectively. In each profile, row i represents the values of the dynamic parameters observed after execution of the DAG for iteration i is finished. Thus, when execution of the task iteration begins, the on-line module does not know the (simulated) actual values of the dynamic parameters for that iteration. The on-line module has to determine a mapping for iteration i based on the dynamic parameter values of iteration $i-1$. The methods for generating the DAGs and execution profiles can be found in [34, 35].

In generating the off-line mapping tables, each dynamic parameter range is partitioned into four equal intervals, creating $4^4 = 256$ regions. Thus, the MDDG Table stores 256 mappings. Ten sample vectors are randomly chosen from each region. Because the GA for each sample vector is performed with 10 different initial populations, the GA is executed $256 \times 10 \times 10 = 25,600$ times to build each MDDG Table. The reconfiguration time is assumed to be 1,000 for these experiments.

6.5.7.4 Results First consider the results of scheduling a 10-node random task DAG using the two execution profiles. The structure and parameters of an example 10-node random DAG are shown in Fig. 6.12. Detailed results of using the four approaches for Profile A are shown in Table 6.2. The definitions of the data columns are (1) $t(\text{map}[i-1])$: the task execution time of iteration i using the mapping chosen at the end of iteration $i-1$, denoted by $\text{map}[i-1]$; (2) $t'(\text{tab}[i-1])$: the task execution time of the mapping stored in the MDDG Table, denoted by $\text{tab}[i-1]$, of the sample vector whose region includes the dynamic parameter values at iteration $i-1$; (3) rc: the reconfiguration time, if remapping is performed; (4) $t(\text{ect}[i-1])$: the execution time of the mapping, denoted by $\text{ect}[i-1]$, determined using the ECT algorithm with the parameters at iteration $i-1$; (5) $t(\text{ga}[i-1])$: the task execution time of iteration i by applying the mapping determined by the GA using the dynamic parameter values from iteration $i-1$; and (6) $t(\text{ga}[i])$: the task execution time of iteration i determined by the GA using the exact dynamic parameter values for iteration i.

As can be seen from Table 6.2, the On–Off approach of dynamically using off-line-derived mappings resulted in much smaller total execution time (1,115,545) compared to that of using the ECT algorithm (1,668,705). The improvement is

(a)

(b)

s_i	a_i	b_i	c_i
s_0	9	24	49
s_1	42	61	9
s_2	42	50	43
s_3	2	34	47
s_4	9	10	38
s_5	33	59	76
s_6	63	45	29
s_7	56	14	54
s_8	48	68	10
s_9	36	25	29

(c)

	d_{ij}	e_{ij}
$s_0 \rightarrow s_1$	5	3
$s_0 \rightarrow s_2$	6	6
$s_0 \rightarrow s_3$	2	1
$s_0 \rightarrow s_4$	7	5
$s_0 \rightarrow s_5$	3	7
$s_0 \rightarrow s_8$	5	6
$s_1 \rightarrow s_7$	3	2
$s_2 \rightarrow s_9$	7	7
$s_4 \rightarrow s_6$	4	2
$s_6 \rightarrow s_7$	5	6
$s_6 \rightarrow s_8$	7	8
$s_6 \rightarrow s_9$	9	10

(d)

	h_{i0}	h_{i1}	h_{i2}	h_{i3}
s_0	0.4897	0.6815	0.7711	0.7503
s_1	0.2828	0.7129	0.4511	0.2725
s_2	0.2575	0.8511	0.5096	0.8779
s_3	0.6337	0.6921	0.8479	0.3451
s_4	0.8283	0.2745	0.4114	0.2836
s_5	0.7267	0.3124	0.2600	0.3354
s_6	0.3932	0.7026	0.8072	0.8066
s_7	0.5276	0.7990	0.5081	0.7942
s_8	0.5876	0.3863	0.6515	0.6472
s_9	0.8760	0.8794	0.6965	0.2407

Figure 6.12 (*a*) An example of a 10-node randomly generated task graph; (*b*) coefficients of the subtask execution-time equations; (*c*) coefficients of the intersubtask communication data equations; (*d*) heterogeneity factors h_{iu} for the subtask execution-time equation.

TABLE 6.2   Results for the 10-Node Random Graph using Profile A

i	On–Off			ECT			GA On-Line	Ideal
	$t(\text{map}[i-1])$	$t'(\text{tab}[i-1])$	rc	$t(\text{map}[i-1])$	$t(\text{ect}[i-1])$	rc	$t(\text{ga}[i-1])$	$t(\text{ga}[i])$
0	—	54,391	1,000	—	74,516	1,000	—	46,228
1	47,508	46,045	1,000	76,384	76,029	0	47,508	47,508
2	47,993	47,286	0	79,227	75,866	1,000	49,095	45,280
3	48,301	51,679	0	75,375	74,586	0	50,306	45,856
4	48,921	51,679	0	76,912	74,755	1,000	48,520	43,611
5	44,035	51,679	0	74,966	71,491	1,000	53,579	40,937
6	49,121	51,679	0	76,226	75,020	1,000	48,410	46,496
7	53,225	52,891	1,000	79,468	77,336	1,000	51,161	47,580
8	55,303	52,891	1,000	81,944	79,878	1,000	55,447	52,809
9	56,355	51,679	0	83,053	82,556	0	52,846	50,539
10	63,288	63,958	0	86,042	87,381	0	55,887	53,733
11	56,631	52,610	1,000	82,874	88,226	0	58,984	51,973
12	58,025	55,142	1,000	87,122	81,860	1,000	63,183	53,733
13	57,410	63,958	0	87,644	87,182	0	57,948	54,311
14	59,774	53,073	1,000	85,449	87,898	0	56,267	52,328
15	58,932	60,754	0	86,152	87,311	0	59,472	55,476
16	55,878	58,723	0	81,710	81,883	0	55,878	55,878
17	58,200	59,774	0	81,434	76,982	1,000	56,981	51,227
18	60,966	63,958	0	88,524	81,581	1,000	56,961	54,483
19	63,071	63,958	0	91,701	84,474	1,000	60,043	59,846
20	65,608	—	—	95,498	93,014	—	58,528	53,842
Total	1,115,545			1,668,705			1,097,004	1,065,040

Note: "Total" below is the total task execution time for 20 iterations.

approximately 33%. The On–Off approach consistently resulted in performance that was comparable to the infeasible GA On-line scheme (about 2% worse), and was only marginally outperformed by the Ideal (but impossible) method (about 5% worse). Indeed, one very interesting observation is that at some iterations (i.e., iterations 2, 3, 5, 8, 11, 12, 13, and 15), the On–Off approach generated a mapping
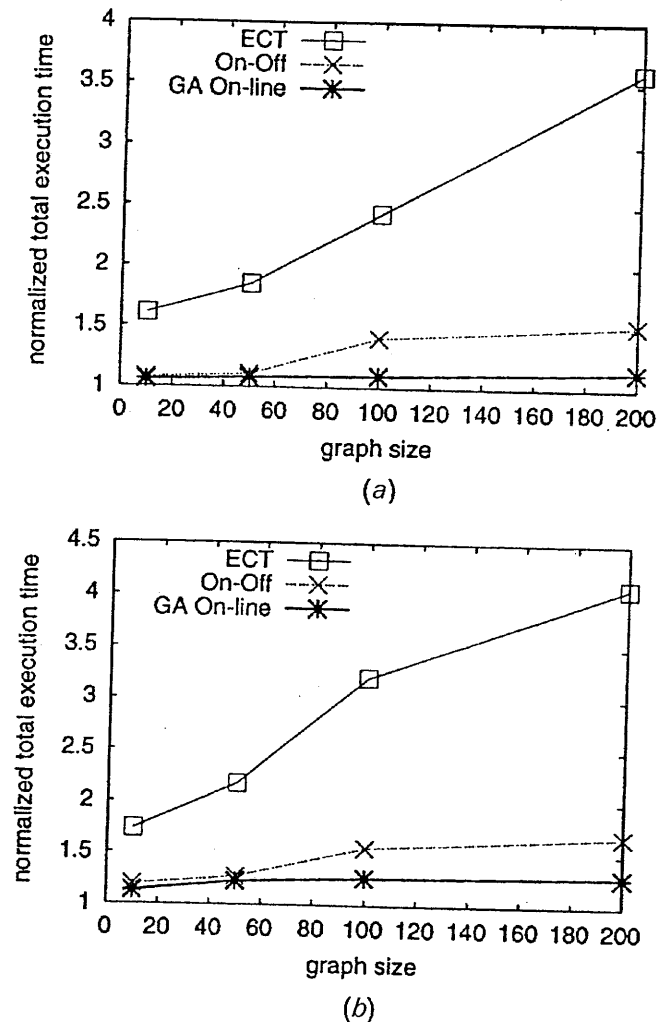

(a)

(b)

Figure 6.13 Comparison of normalized total task execution times for randomly structured graphs with (a) Profile A and (b) Profile B.

resulting in a shorter execution time than the GA On-line approach. Thus, the GA On-line approach, because it computes a mapping optimized for the specific dynamic parameter values from the previous iteration, is sometimes not as robust to changes in the dynamic parameter values as the region-sampling techniques used by the On–Off approach.

Figure 6.13a and b show the average *normalized total execution times* of the ECT, On–Off, and GA On-line approaches with respect to the Ideal method for the randomly structured graphs. The normalized total execution time of each test case is calculated by dividing the total execution time of a particular approach (e.g., ECT) by that of the Ideal method. Each point on the curves gives the average value of 10 test cases. As can be seen, the normalized total execution times are, in general, slightly higher for Profile B than for Profile A. The normalized total execution times of On–Off and GA On-line are consistently of similar values for all graph sizes. However, the ECT approach performed much worse, especially for large graphs (sizes 100 and 200). An explanation for this phenomenon is because the ECT algorithm employs a strictly greedy scheduling method, the effect of making mistakes at early stages of scheduling can be propagated to later stages. The adverse impact of such a greedy approach can be more profound for larger graphs. For more detailed results, the reader is referred to [34, 35].

6.5.8 Summary

This study focused on the design of a semistatic approach for using GA derived mappings in real time. For the computational environment considered, an HC Kernel was presented for making real-time, on-line, input-data-dependent remappings of the application subtasks to the processors in the heterogeneous parallel hardware platform using previously stored off-line, statically determined mappings. In particular, it was shown how the HC Kernel can be used to create the MDDG Table off-line using a GA with a novel dynamic parameter space partitioning and sampling technique, and then use it to make real-time, on-line decisions and selections of mappings. The simulation results indicated that the semistatic On–Off approach is effective in that it consistently outperformed a fast dynamic mapping heuristic by a considerable margin, and gave reasonable performance even when compared to the impossible approach of using the genetic algorithm on-line with future knowledge of the next iteration's dynamic parameters. The On–Off approach reviewed here can also be used for other application domains and classes of hardware platforms whose characteristics are similar to those of the applications and platforms considered here.

6.6 STATIC MATCHING AND SCHEDULING FOR META-TASKS

6.6.1 Introduction

This study implemented 11 different static meta-task mapping heuristics, including 2 evolutionary approaches, so a comparison could be made of their performance using

a common simulated HC environment [6]. Section 6.6.2 presents information about how the ETC matrices were generated. Descriptions of the 11 heuristics implemented appear in Section 6.6.3. Last, a sampling of results from the experiments are shown in Section 6.6.4.

6.6.2. ETC Matrices

For the simulation studies, characteristics of the ETC matrices were varied in an attempt to represent a range of possible HC environments. The ETC matrices used were generated using the following method. Initially, a $|T| \times 1$ *baseline column vector*, B, of floating-point values is created. Let ϕ_b be the upper bound of the range of possible values within the baseline vector. The baseline column vector is generated by repeatedly selecting a uniform random number, $x_b^i \in [1, \phi_b)$, and $B(i) = x_b^i$ for $0 \le i < |T|$. Next, the rows of the ETC matrix are constructed. Each element ETC(i, j) in row i of the ETC matrix is created by taking the baseline value, $B(i)$, and multiplying it by a uniform random number, $x_r^{i,j}$, which has an upper bound of ϕ_r. This new random number, $x_r^{i,j} \in [1, \phi_r)$, is called a *row multiplier*. One row requires $|M|$ different row multipliers, $0 \le j < |M|$. Each row i of the ETC matrix then can be described as ETC$(i, j) = B(i) \times x_r^{i,j}$, for $0 \le j < |M|$. The baseline column itself does not appear in the final ETC matrix.) This process is repeated for each row until the $|M| \times |T|$ ETC matrix is full. Therefore any given value in the ETC matrix is within the range $[1, \phi_b \times \phi_r)$.

To evaluate the heuristics for different mapping scenarios, the characteristics of the ETC matrix were varied based on several different methods from [3]. The amount of variance among the execution times of tasks in the meta-task for a given machine is defined as *task heterogeneity*. Task heterogeneity was varied by changing the upper bound of the random numbers within the baseline column vector. *High* task heterogeneity was represented by $\phi_b = 3,000$, and *low* task heterogeneity by $\phi_b = 100$. *Machine heterogeneity* represents the variation that is possible among the execution times for a given task across all the machines. Machine heterogeneity was varied by changing the upper bound of the random numbers used to multiply the baseline values. *High* machine heterogeneity values were generated using $\phi_r = 1,000$, while *low* machine heterogeneity values used $\phi_r = 10$. These heterogeneous ranges are based on one type of expected environment for MSHN. The ranges were chosen to reflect the fact that in real situations there is more variability across task execution times on a given machine than the execution time for a single task across different machines.

To further vary the ETC matrix in an attempt to capture more aspects of realistic mapping situations, different ETC matrix consistencies were used. An ETC matrix is said to be *consistent* if whenever a machine j executes any task i faster than machine k, then machine j executes all tasks faster than machine k [3]. Consistent matrices were generated by sorting each row of the ETC matrix independently. In contrast, *inconsistent* matrices characterize the situation where machine j is faster than machine k for some tasks, and slower for others. These matrices are left in the unordered, random state in which they were generated. *Semiconsistent* matrices are

TABLE 6.3 Sample 8×8 Excerpt from an ETC Matrix with Inconsistent, High-Task, High-Machine Heterogeniety

	Machines							
	436,735	815,309	891,469	1,722,197	1,340,988	740,028	1,749,673	251,140
	950,470	933,830	2,156,144	2,202,018	2,286,210	2,779,669	220,536	1,769,184
	453,126	479,091	150,324	386,338	401,682	218,826	242,699	11,392
Tasks	1,289,078	1,400,308	2,378,363	2,458,087	351,387	925,070	2,097,914	1,206,158
	646,129	576,144	1,475,908	424,448	576,238	223,453	256,804	88,737
	1,061,682	43,439	1,355,855	1,736,937	1,624,942	2,070,705	1,977,650	1,066,470
	10,783	7,453	3,454	23,720	29,817	1,143	44,249	5,039
	1,940,704	1,682,338	1,978,545	788,342	1,192,052	1,922,914	701,336	1,052,728

inconsistent matrices that include a consistent submatrix. For the semiconsistent matrices used here, the row elements in column positions $\{0, 2, 4, \ldots\}$ of row i are extracted sorted, and replaced in order, while the row elements in column positions $\{1, 3, 5, \ldots\}$ remain unordered. (That is, the even columns are consistent and the odd columns are, in general, inconsistent.)

Sample ETC matrices are shown in Tables 6.3 and 6.4. All results in this study used ETC matrices that were of size $|T| = 512$ tasks by $|M| = 16$ machines. While it was necessary to select some specific parameter values to allow implementation of a simulation, the characteristics and techniques presented here are completely general. Therefore, if these parameter values do not apply to a specific situation of interest, researchers may use other ranges, distributions, matrix sizes, etc.

4.6.3 Description of Heuristics

The definitions of the 11 static meta-task mapping heuristics are provided below. First, some preliminary terms must be defined. *Machine availability time,* avail(j), is the earliest time a machine j can complete the execution of all the tasks that previously have been assigned to it. The *completion time* (ct) for a new task i on machine j is ct(i, j) or the machine availability time plus the execution time of task i.

TABLE 6.4 Sample 8×8 Excerpt from an ETC Matrix with Inconsistent, Low-Task, Low-Machine Heterogeneity

	Machines							
	512	268	924	494	611	606	921	209
	8	16	23	19	27	22	19	8
	228	238	107	180	334	88	192	125
	345	642	136	206	559	349	640	664
Tasks	117	235	149	71	136	363	182	359
	240	412	259	319	237	338	178	537
	462	93	574	449	421	559	487	298
	119	36	224	194	176	156	182	192

on machine j, i.e., $ct(i, j) = avail(j) + ETC(i, j)$. The performance criterion used to compare the results of the heuristics is the maximum value of $ct(i, j)$, for $0 \le i < |T|$ and $0 \le j < |M|$, for each heuristic, also known as the *makespan* [39]. Each heuristic is attempting to minimize the makespan (i.e., finish execution of the meta-task as soon as possible).

The descriptions below implicitly assume that the machine availability times are updated after each task is mapped. For cases when tasks can be considered in an arbitrary order, the order in which the tasks appeared in the ETC matrix was used. Some of the heuristics listed below had to be modified from their original implementation to better handle the environment under consideration.

For many of the heuristics, there are control parameter values and/or control-function specifications that can be selected for a given implementation. For the studies here, such values and specifications were selected based on experimentation and/or information in the literature. A more thorough description of each of the heuristics and some of the intuition behind them, along with a listing of some alternative implementations, can be found in [6].

OLB Opportunistic load balancing (OLB) assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected execution time on that machine [2, 21, 22].

UDA In contrast to OLB, *user-directed assignment* (UDA) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability. UDA is sometimes referred to as limited best assignment (LBA), as in [2, 21]. In general, this heuristic is obviously not applicable to HC environments characterized by consistent ETC matrices.

Fast Greedy Fast Greedy assigns each task, in arbitrary order, to the machine with the minimum completion time for that task [2].

Min–min The *Min–min* heuristic begins with the set U of all unmapped tasks. Then, the set of *minimum completion times*, $MCT = \{mct_i : mct_i = \min_{0 \le j < |M|}(ct(i,j)), \text{ for each } i \in U\}$, is found. Next, the task from U with the overall *minimum* completion time is selected and assigned to the corresponding machine (hence the name Min–min). Last, the newly mapped task is removed from U, and the process repeats until tasks are mapped (i.e., U is empty) [2, 21, 29].

Max–min The *Max–min* heuristic is very similar to Min–min. The Max–min heuristic also begins with the set U of all unmapped tasks. Then, the set of MCTs is found. Next, the task from U with the overall *maximum* completion time is selected and assigned to the corresponding machine (hence the name Max–min). Last, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., U is empty) [2, 21, 29].

Greedy The *Greedy* heuristic is literally a combination of the Min–min and Max–min heuristics. The Greedy heuristic performs both of the Min–min and Max–min heuristics, and uses the better solution [2, 21].

GA The *genetic algorithm* (GA) implemented in this study was adapted from [58] (see Section 6.4 for a description of [58]) to be applied to meta-tasks, and unless otherwise noted, uses similar values and techniques. The GA operates on a population of 200 chromosomes for a given meta-task. Each chromosome is a $|T| \times 1$ vector, where position i ($0 \le i < |T|$) represents task i, and the entry in position i is the machine to which the task has been mapped. The makespan is the fitness value. The initial population is generated using two methods: (1) 200 randomly generated chromosomes from a uniform distribution, or (2) one chromosome (seed) that is the Min–min solution and 199 random solutions (mappings). The probability of crossover was 60% and mutation was 40%. The stopping criteria that usually occurred in testing were "no changes in the elite chromosome in 150 iterations". Eight GA runs were performed (four times with different initial populations from each method), and the best of the eight mappings is used as the final solution

SA Simulated annealing (SA) is an iterative technique that considers only one possible solution (mapping) for each meta-task at a time. This technique uses the same representation for a solution as the chromosome for the GA. Mutations of the current chromosome are also performed similarly to the GA.

The corresponding selection process for SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space (e.g., [13, 33, 42]). This probability is based on a *system temperature* that decreases for each iteration. As the system temperature "cools," it is more difficult for currently poorer solutions to be accepted. The initial system temperature is the makespan of the initial (random) mapping.

Therefore, the SA begins with the current chromosome, mutates it, and then uses the system temperature to determine whether to accept or reject this new solution. After each mutation, the system temperature is decreased by 10%. This represents one iteration of SA. The heuristic stops when there is no change in the makespan of the solution for 150 iterations or the system temperature reaches zero.

GSA The *genetic simulated annealing* (GSA) heuristic is a combination of the GA and SA techniques [10, 45]. In general, GSA follows procedures similar to the GA just outlined. GSA operates on a population of 200 chromosomes, uses a Min–min seed in four out of eight initial populations, and performs similar mutation and crossover operations. However, for the selection process, GSA uses the SA cooling schedule and system temperature, and a simplified SA decision process for accepting or rejecting new chromosomes. GSA also employs elitism to guarantee that the best solution always remains in the population.

Tabu Tabu search is a solution space search that keeps track of the regions of the solution space that have already been searched, so as not to repeat a search near these areas [15, 24]. A solution (mapping) uses the same representation as a chromosome in the GA approach.

The implementation of Tabu search used here begins with a random mapping, generated from a uniform distribution. Starting with the first task in the mapping, task $i = 0$, each possible pair of tasks is formed, (i, j) for $0 \leq i < |T| - 1$ and $i < j < |T|$. As each pair of tasks is formed, they potentially exchange machine assignments. This constitutes a *short hop*. After each exchange, the new makespan is evaluated. If the new makespan is an improvement, the new exchange is retained, forming a new mapping (a *successful short hop*). New short hops are generated until a maximum number of successful hops have been made (see next paragraph) or all combinations of task pairs have been exhausted with no further improvement.

At this point, the final mapping from the local solution space search is added to the *tabu list*. Next, a new random mapping is generated, and it must differ from each mapping in the tabu list by at least half of the machine assignments (a *successful long hop*). Then the short hops are repeated. The final stopping criterion for the heuristic is a total of 1200 successful hops (short and long combined). Then, the best mapping from the tabu list is the final answer.

A* A* has been applied to many other task-allocation problems (e.g., [11, 31, 42, 44]). The technique used here is similar to [11].

A* is a tree search beginning at a root node that is a null solution. As the tree grows, intermediate nodes represent partial solutions (a subset of tasks are assigned to machines). The partial solution of a child node has one more task mapped than the parent node. Call this additional task a. Each parent node generates $|M|$ children, one for each possible mapping of a. Based on experimentation and a desire to keep execution time of the heuristic tractable the maximum number of leaf nodes in the tree at any one time is limited in this study to $n_{\max} = 1,024$.

Each node, n, has a *cost function*, $f(n)$, associated with it. The cost function is an estimated lower bound on the makespan of the best solution, which includes the partial solution represented by node n. (The lower bound on the time for executing the remaining tasks includes the assumptions that each task is assigned to its preferred machine and that all machines are equally utilized.)

Thus, beginning with the root, the node with the minimum $f(n)$ is expanded by its $|M|$ children, until $n_{\max}$ leaf nodes are created. From that point on, any time a node is added, the tree is pruned by deleting the leaf node with the largest $f(n)$. This process continues until a leaf node representing a complete mapping is reached. Note that if the tree is not pruned, this method is equivalent to an exhaustive search.

6.6.4 Results

An interactive software tool has been developed that allows simulation, testing, and demonstration of the heuristics examined in Section 6.6.3. This software tool operates on the meta-tasks defined by the ETC matrices described in Section 6.6.2.

The software allows a user to specify $|T|$ and $|M|$, to select which types of ETC matrices to use, and to choose which heuristics to execute. It then generates the specified ETC matrices, executes the desired heuristics, and displays the results, similar to Fig. 6.14. The results discussed in this section were generated using portions of this software.

When comparing mapping heuristics, the execution time of the heuristics themselves is an important consideration. For the heuristics listed, the execution times varied greatly. The experimental results discussed below were obtained on a Pentium II 400-MHz processor with 1 GB of RAM. Each of the simpler heuristics [OLB, UDA, Fast Greedy, and Greedy (which includes both Min–min and Max–min)] executed in a few seconds for one ETC matrix with $|T| = 512$ and $|M| = 16$. For the same-sized ETC matrix, SA and Tabu, both of which manipulate a single solution during an iteration, averaged less than 30 s. GA and GSA required approximately 60 s per matrix because they manipulate entire populations, and A* required about 20 min per matrix.

The resulting meta-task execution times (makespans) from the simulations of sample HC environments are shown in Fig. 6.14. All experimental results represent the execution time of a meta-task (defined by a particular ETC matrix) based on the mapping found by the heuristic specified, averaged over 100 different ETC matrices of the same type (i.e., 100 mappings). For each heuristic, the range bars show the minimum and maximum meta-task execution times over the 100 mappings (100 ETC matrices) used to compute the average meta-task execution time.

For the four consistent cases (i.e., each combination of high and low task and machine heterogeneity), the UDA algorithm mapped all tasks to the same machine, resulting in the worst performance by an order of magnitude (therefore, UDA is not included in Fig. 6.14a). GA performed the best for the consistent cases. This was due in large part to the good performance of the Min–min heuristic. The best GA solution always came from one of the populations that had been seeded with the Min–min solution. As is apparent in the figure, Min–min performed very well on its own, giving the second best results. However, the mutation, crossover, and selection operations of the GA were always able to improve on this solution. GSA, which also used a Min–min seed, did not always improve upon the Min–min solution. Because of the probabilistic procedure used during selection, GSA would sometimes accept poorer intermediate solutions. These poorer intermediate solutions never led to better final solutions; thus, GSA gave poorer results than the GA. The performance of A* was hindered because the estimates made by $f(n)$ are not as accurate for consistent cases as they are for inconsistent and semiconsistent cases.

These results suggest that if the best overall solution is desired, the GA should be employed. However, the improvement of the GA solution over the Min–min solution was never more than 10%. Therefore, the Min–min heuristic may be more appropriate in certain situations, given the difference in execution times of the two heuristics.

For the four inconsistent test cases, UDA performs very well while the performance of OLB degrades. The OLB performance degradation for the inconsistent
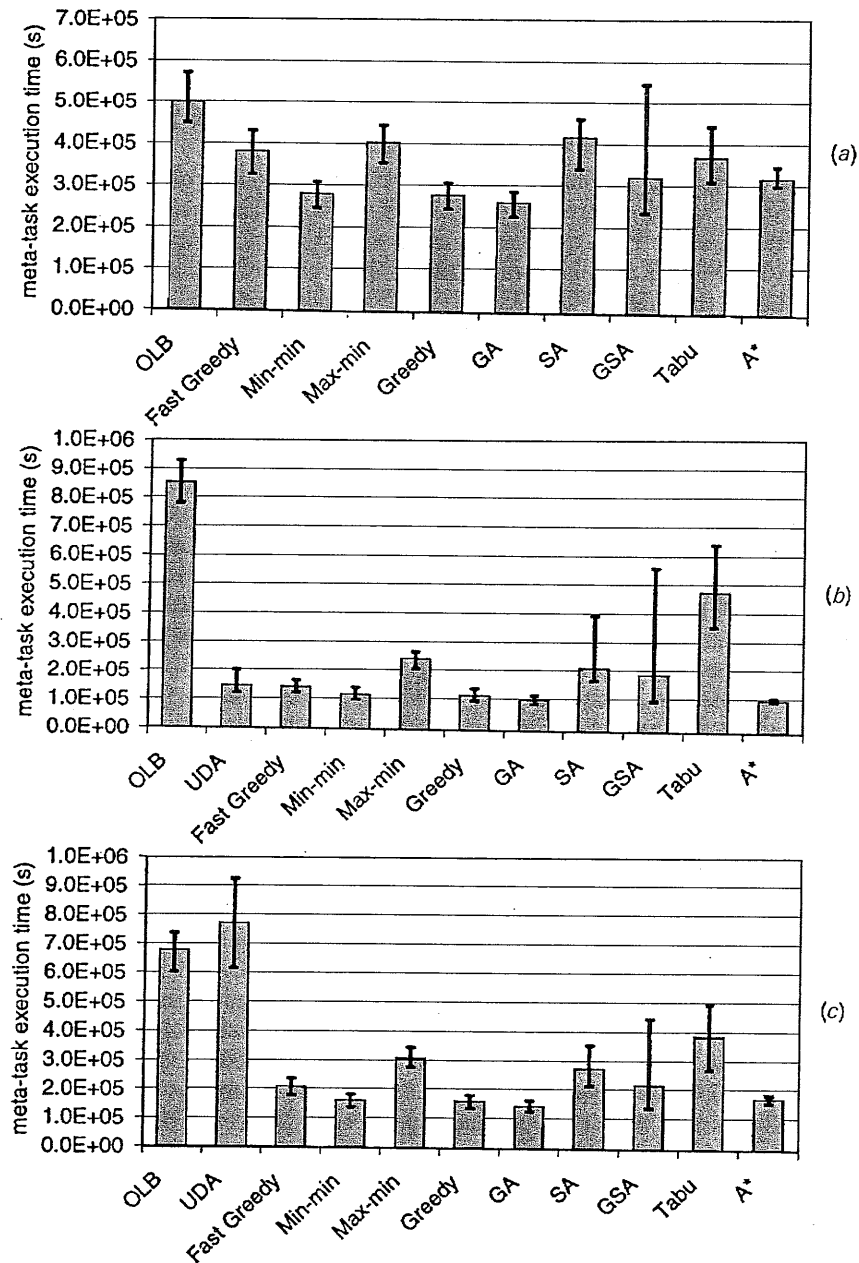
Figure 6.14 Meta-task mapping results for 512 low-heterogeneity tasks and 16 high-heterogeneity machines for ETCs that are (*a*) consistent, (*b*) inconsistent, and (*c*) semiconsistent. The graphs show the mean and range over 100 trials.

cases can likely be attributed to more "unfavorable" assignments occurring as compared to the consistent cases. For example, for the consistent cases, one machine executes all of the tasks quickest, so more tasks will be assigned to this machine, which is again the best assignment for that task. This phenomenon is less likely to occur for the inconsistent cases because there is no one "best" machine for all of the tasks. In contrast, UDA improves because the "best" machines are distributed across the set of machines, thus task assignments will be more evenly distributed among the set of machines, avoiding load imbalance (to some extent, this is due to the random distributions used in the simulation). Similarly, Fast Greedy and Min–min performed very well, and slightly outperformed UDA, because the machines providing the best task completion times are more evenly distributed among the set of machines. Min–min was also better than Max–min for all of the inconsistent cases. The advantages Min–min gains by mapping "best-case" tasks first outweighs the possible savings in "packing" that Max–min has by mapping "worst-case" tasks first [6].

Tabu gave the second poorest results for the inconsistent cases, with makespans that were always at least 20% worse than Max–min (the third poorest heuristic in terms of mean performance). Inconsistent matrices generated more successful short hops than the associated consistent matrices. Therefore, fewer long hops were generated and less of the solution space was searched, resulting in poorer solutions.

GA and A* had the best average makespans, and were usually within a small constant factor of each other. GA again benefited from having the initial Min–min mapping. A* did well because, if the tasks get more evenly distributed among the machines, this more closely matches the lower-bound estimates $f(n)$.

For semiconsistent cases with high machine heterogeneity, the UDA heuristic again gave the worst results. Intuitively, UDA is suffering from the same problem as in the consistent cases: at least half of all tasks are getting assigned to the same machine. OLB does poorly for high machine heterogeneity cases because worst-case matchings will have higher execution times for high machine heterogeneity. For low machine heterogeneity, the worst-case matchings have a much lower penalty. The best heuristics for the semiconsistent cases were Min–min and GA. This is not surprising because these were two of the best heuristics from the consistent and inconsistent tests, and semiconsistent matrices are a combination of consistent and inconsistent matrices. Min–min was able to do well because it searched the entire row for each task and assigned a high percentage of tasks to their first-choice machine. GA was robust enough to handle the consistent components of the matrices, and did well for the same reason mentioned for inconsistent matrices.

6.6.5 Summary

The goal of this study was to provide insights and a basis for comparison of 11 different heuristics for the mapping of static meta-tasks in different HC environments. The characteristics of the ETC matrices used as input for the heuristics and the methods used to generate them were specified. The implementation of a collection of 11 heuristics from the literature was described. The results of the mapping heuristics were discussed, revealing the best heuristics to use in certain

environments. For the situations, implementations, and parameter values used here, GA was the best heuristic for most cases, followed closely by Min–min, with A* also doing well for inconsistent matrices. The comparisons in this study can be used by researchers as a baseline for evaluating the efficacy of new techniques.

6.7 SUMMARY

In a mixed-machine, distributed, heterogeneous computing environment, there is a suite of high-performance machines with different computational capabilities. These machines are interconnected by high-speed links. Such a suite of machines can be used to execute a single application, whose subtasks have diverse execution requirements, or to execute a meta-task, which is a collection of independent tasks with different computational needs.

For the single-application case, subtasks are assigned to and executed on the machines that will result in a minimal execution time for the overall task, considering subtask computation time and intermachine communication overhead. A GA-based approach was described for matching subtasks to machines and scheduling the execution of the subtasks. This approach is used off-line and is based on expected computation and communication times for the subtasks. A dynamic-parameter-sampling-based method for using this approach on-line in certain application domains was also presented. This on-line adaptation may be helpful when computation and communication times can vary significantly from the expected values, depending on the input data being processed.

For the meta-task case, the goal is to match each task to a machine in the suite so that the execution time for the entire meta-task is minimized. In the situation considered here, the matching process occurs off-line and plans the machine assignments and execution schedule for a meta-task for a later time interval (e.g., a large set of production jobs that will execute the next day). In this situation, the matching process can also be used to determine if a proposed set of machines can perform the meta-task within some time limit. The use of GA-based approach for this environment was discussed.

In summary, much work has been done using GAs of various types to solve the problem of matching and scheduling of tasks and meta-tasks in a mixed-machine, distributed, heterogeneous computing environment. This chapter has discussed three types of genetic algorithms that have been studied to solve this problem. Each implementation has particular specifications and qualifications that were being met. In all cases, the genetic algorithm proved to be a useful method for solving the making and scheduling problem being researched.

ACKNOWLEDGMENTS

REFERENCES

1. H. M. Alnuweiri and V. K. Prasanna, Parallel architectures and algorithms for image component labeling. *IEEE Trans. Pattern Anal. Machine Intelligence*, 14(10):1014–1034, Oct. 1992.

2. R. Armstrong, D. Heusgen, and T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proceedings of the Seventh IEEE Heterogeneous Computing Workshop (HCW '98)*, pp. 79–87, Mar. 1998.

3. R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance*. Thesis, Department of Computer Science, Naval Postgraduate School Monterey, CA, Sept. 1997.

4. P. Baglietto, M. Maresca, M. Migliardi, and N. Zingirian, Image processing on high-performance RISC systems. *Proc. IEEE,* 84(7):917–930, July 1996.

5. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Proceedings of the Seventh IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 330–335, Oct. 1998.

6. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing system. In *Proceedings of the Eighth IEEE Workshop on Heterogeneous Computing Systems (HCW '99)*, pp. 15–29, Apr. 1999.

7. J. R. Budenske, R. S. Ramanujan, and H. J. Siegel, Modeling ATR applications for intelligent execution upon a heterogeneous computing platform. In *Proceedings of 1997 IEEE International Conference on Systems, Man and Cybernetics*, pp. 649–656, Oct. 1997.

8. J. R. Budenske, R. S. Ramanujan, and H. J. Siegel, A method for the on-line use of off-line derived remappings of iterative automatic target recognition tasks onto a particular class of heterogeneous parallel platforms. *Jour. Supercomputing*, 12(4):387–406, Oct. 1998.

9. E. A. Carmona and M. D. Rice, Modeling the serial and parallel fractions of a parallel program. *Parallel and Distributed Computing,* 13(3):286–298, Nov. 1991.

10. H. Chen, N. S. Flann, and D. W. Watson, Parallel genetic simulated annealing: A massively parallel SIMD approach, *IEEE Trans. Parallel and Distributed Computing*, 9(2):126–136, Feb. 1998.

11. K. Chow and B. Liu, On mapping signal processing algorithms to a heterogeneous multiprocessor system. In *1991 International Conference on Acoustics, Speech, and Signal Processing (ICASSP '91)*, Vol. 3, pp. 1585–1588, May 1991.

12. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge University Press, Cambridge, MA, 1992.

13. M. Coli and P. Palazzari, Real time pipelined system design through simulated annealing. *Jour. Sys. Architecture*, 42(6–7):465–475, Dec. 1996.

14. L. Davis, ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

15. I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, Improving search by incorporating evolution principles in parallel tabu search. In *1994 IEEE Conference on Evolutionary Computation*, Vol. 2, pp. 823–828, 1994.

16. T. H. Einstein, Mercury computer systems' modular heterogeneous RACE multicomputer. In *Proceedings of the Sixth IEEE Heterogeneous Computing Workshop (HCW '97)*, pp. 60–71, April 1997.

17. M. M. Eshaghian, ed., *Heterogeneous Computing*. Artech House, Northwood, MA, 1996.

18. M. M. Eshaghian and M. E Shaaban, Cluster-M programming paradigm. *Int. Jour. High Speed Computing*, 6(2):287–309, June 1994.

19. D. Fernandez-Baca, Allocating modules to processors in a distributed system. *IEEE Trans. Software Eng.* SE-15(11):1427–1436, Nov. 1989.

20. R. F. Freund, Optimal selection theory for superconcurrency. *Supercomputing '89*, pp. 699–703, Nov. 1989.

21. R. F. Freund, M. Gherrity, S. Ambrosius. M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, Scheduling resources in multi-user heterogeneous, computing environments with Smart-Net. In *Proceedings of the Seventh IEEE Heterogeneous Computing Workshop (HCW '98)*, pp. 184–199, March 1998.

22. R. F. Freund and H. J. Siegel, Heterogeneous processing, *IEEE Comuputer*, 26(6):13–17, June 1993.

23. A. Ghafoor and J. Yang, Distributed heterogeneous supercomputing management system. *IEEE Computer*, 26(6):78–86, June 1993.

24. F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, MA, 1997.

25. D. B. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

26. D. A. Hensgen, T. Kidd, M. C. Schnaidt, D. St. John, H. J. Siegel, T. D. Braun, M. Maheswaran, S. Ali, J.-K. Kim, C. Irvine, T. Levin, R. Wright, R. F. Freund, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, An overview of MSHN: A management system for heterogeneous networks. In *Proceedings of the Eighth IEEE Workshop on Heterogeneous Computing Systems (HCW '99)*, pp. 184–198, April 1999.

27. J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

28. R. Hoebelheinrich and R. Thomsen, Multiple crossbar network integrated supercomputing framework, *Supercomputing '89*, pp. 713–720, Nov. 1989.

29. O. H. Ibarra and C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Jour. ACM*, 24(2):280–289, April 1977.

30. M. A. Iverson, F. Ozguncr, and G. J. Follen, Parallelizing existing applications in a distributed heterogeneous environment. In *Proceedings of the Fifth IEEE Heterogeneous Computing Workshop (HCW '95)*, pp. 93–100, April 1995.

31. M. Kafil and I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3):42–51, July–Sept. 1998.

32. A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

33. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

34. Y.-K. Kwok, A. A. Maciejewski, H. J. Siegel, A. Ghafoor, and I. Ahmad, Evaluation of a semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. In *Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pp. 204–209, June 1999.

35. Y.-K. Kwok, A. A. Maciejewski, H. J. Siegel, A. Ghafoor, and I. Ahmad, *Implementation and Perforrnance Study of a Semi-static Approach to Mapping Dynamic Iterative Tasks onto Heterogeneous Computing Systems*. Technial report HKUST-CS99-15, Department of Computer Science, The Hong Kong University of Science and Technology, in preparation.

36. M. Maheswaran, T. D. Braun, and H. .J. Siegel, Heterogeneous distributed computing. In J. Webster, ed., *Encyclopedia of Electrical Engineering and Electronics*, Vol. 8, pp. 679–690, Wiley, New York, 2000.

37. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Jour. Parallel and Distributed Computing*, 59(2):107–121, Nov. 1999.

38. B. Narahari, A. Youssef, and H. A. Choi, Matching and scheduling in a generalized optimal selection theory. In *Proceedings of the Third IEEE Heterogeneous Compuing Workshop (HCW '94)*, pp. 3–8, April 1994.

39. M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

40. J. L. Ribeiro Filho and P. C. Treleaven, Genetic-algorithm programming environments. *IEEE Computer*, 27(6):28–43, June 1994.

41. G. Rudolph, Convergence analysis of canonical genetic algorithms. *IEEE Trans. Neural Networks*, 5(1):96–101, Jan. 1994.

42. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

43. K. C. Sevcik, Characterizations of parallelism in applications and their use in scheduling. *Performance Evaluation Rev.*, 17(1):171–180, May 1989.

44. C.-C Shen and W.-H. Tsai, A graph matching approach to optimal task assignment in distributed computing system using a minmax criterion. *IEEE Trans. Computers*, C-34(3):197–203, March 1985.

45. P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *Proceedings of the Fifth IEEE Heterogeneous Computing Workshop (HCW '96)*, pp. 98–104, April 1996.

46. H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and L. A. Li, Heterogeneous computing. In A. Y. Zomaya ed., *Parallel and Distributed Computing Handbook*, pp. 725–761, McGraw-Hill, New York, 1996.

47. H. J. Siegel, H. G. Dietz and J. K. Antonio, Software support for heterogeneous computing. In A. B. Tucker Jr. Ed., *The Computer Science and Engineering Handbook*, pp. 1886–1909, CRC Press, Boca Raton, FL, 1997.

48. H. Singh and A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proceedings of the Fifth IEEE Heterogeneous Computing Workshop (HCW '96)*, pp. 86–97, April 1996.

49. M. Srinivas and L. M. Patnaik, Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, June 1994.

50. V. S. Sunderam, Design issues in heterogeneous network computing. In *IEEE Workshop on Heterogeneous Processing (rev. Ed.)*, pp. 101–112, March 1992.

51. M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Trans. Parallel and Distributed Sys.*, 8(8):857–871, Aug. 1997.

52. Y. G. Tirat-Gefen and A. C. Parker, MEGA An approach to system-level design of application-specific heterogeneous multiprocessors. In *Proceedings of the Fourth IEEE Heterogeneous Computing Workshop (HCW '96)*, pp. 105–117, April 1996.

53. D. Tolmie and J. Renwick, HiPPI: Simplicity yields success. *IEEE Network*, 7(1):28–32, Jan. 1993.

54. J. G. Verly and R. I. Delanoy, Model-based automatic target recognition (ATR) system for forwardlooking groundbased and airborne imaging laser radars (LADAR). *Proc. IEEE*, 84(2): 126–163, Feb. 1996.

55. D. W. Watson, J. K. Antonio, H. J. Siegel, and M. J. Atallah, Static program decomposition among machines in an SIMD/SPMD heterogeneous environment with non-constant mode switching cost. In *Proceedings of the Third IEEE Heterogeneous Computing Workshop (HCW '94)*, pp. 58–65, April 1994.

56. D. W. Watson, J. K. Antonio, H. J. Siegel, R. Gupta, and M. J. Atallah, Static matching of ordered program segments to dedicated machines in a heterogeneous computing environment. In *Proceedings of the Fifth IEEE Heterogeneous Computing Workshop (HCW '96)*, pp. 24–37, April 1996.

57. Q. Wang and K. H. Cheng, List scheduling of parallel tasks. *Information Processing Lett.*, 37(5):291–297, March 1991.

58. L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Jour. Parallel and Distributed Computing*, 47(1):1–15, Nov. 1997.

59. C. C. Weems, G. E. Weaver, and S. G. Dropsho, Linguistic support for heterogeneous parallel processing: A survey and an approach. In *Proceedings of the Third IEEE Heterogeneous Computing Workshop (HCW '94)*, pp. 81–88, April 1994.

60. D. Whitley, The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *1989 International Conference on Genetic Algorithms*, pp. 116–121, Morgan Kaufmann, June 1989.

61. A. Zomaya, C. Ward, and B. Macey, Genetic scheduling for parallel processor systems: Comparative studies and performance issues, *IEEE Trans. Parallel and Distributed Sys.*, 10(8):795–812, Aug. 1999.

Evolving Cellular Automata-Based Algorithms for Multiprocessor Scheduling

F. SEREDYNSKI

Polish Academy of Sciences

7.1   INTRODUCTION

Constructing efficient algorithms for scheduling tasks of a parallel program in a multiprocessor architecture remains an unsolved issue in parallel computing. The problem of multiprocessor scheduling, even limited to a two-processor systems but any parallel program, is known to be NP complete [4, 11]. Despite many studies, progress in this area due to an NP completeness of a scheduling problem is slow and insufficient to automatize corresponding functions of operating systems designed for multiprocessor systems.

Current works [1, 4, 11, 16] concerning the scheduling problem are focused either on selecting problems for which *exact* solutions can be constructed or designing *heuristic* algorithms to find near-optimal solutions for more general cases. In particular, scheduling algorithms based on applying techniques derived from nature, such as simulated annealing, genetic algorithms, or neural networks [6, 13, 15], belong to the latter area of research. While most of the proposed scheduling algorithms are *sequential algorithms*, a new direction in this area is developing *parallel* and *distributed* scheduling algorithms [2, 20].

In this chapter I develop parallel and distributed scheduling algorithms using a recent and very promising technique based on combining evolutionary computation and cellular automata (CAs). CAs present a distributed system of single, locally interacting units that are able to produce global behavior. CAs can be considered a model of naturally existing systems produced by natural evolution. Natural evolution has produced such systems that are capable of producing globally coordinated information processing, unguided by any global criterion or central control. The