TASK ASSIGNMENT HEURISTICS FOR PARALLEL AND DISTRIBUTED CFD APPLICATIONS

N. LOPEZ-BENITEZ*, M. J. DJOMEHRI†, AND R. BISWAS‡

Abstract. This paper proposes a task graph (TG) model to represent a single discrete step of multi-block overset grid computational fluid dynamics (CFD) applications. The TG model is then used to not only balance the computational workload across the overset grids but also to reduce inter-grid communication costs. We have developed a set of task assignment heuristics based on the constraints inherent in this class of CFD problems. Two basic assignments, the smallest task first (STF) and the largest task first (LTF), are first presented. They are then systematically enhanced by integrating the status of the processing units and the interprocessor communication costs. To predict the performance of the proposed task assignment heuristics, extensive performance evaluations are conducted on a synthetic TG with tasks defined in terms of the number of grid points in predetermined overlapping grids. A TG derived from a realistic problem with eight million grid points is also used as a test case.

Key words. Overset grids, task graphs, performance prediction, parallel processing, high performance computing.

AMS subject classifications. 05C90, 68U01, 65Y05

1. Introduction. The availability of massively parallel computational resources poses a challenge to the development of efficient algorithms for high-performance scientific computing. A possible application is the high-fidelity solution of Navier-Stokes equations to predict aerodynamic flow characteristics around complex aerospace configurations. High-end supercomputing is required to reduce the turn-around time of such computational fluid dynamics (CFD) simulations. Multiple processors may be either tightly coupled or geographically separated and networked into a single virtual supercomputer. Seamless access to distributed resources is enabled by metacomputing toolkits such as Globus [7, 8]. Due to the inherent cost effectiveness of aggregated computing, the distributed approach has attracted significant interest and become a research priority in recent years. It has also been the main driver behind the development of NASA's Information Power Grid (IPG) [9].

To handle complex geometric configurations, NASA's CFD production code called OVERFLOW [3] decomposes the flow domain into a union of overset structured grids (also referred to as zones), each of which covers a relatively simple region of the domain. In the parallel implementation, a bin-packing strategy is used to cluster individual grids into groups, where the number of groups is equal to the number of processors. To avoid poor volumetric load balance, the larger grids can be further partitioned into subgrids. As a result, the number of grids and subgrids (collectively known as tasks in this paper) easily exceed the total number of available processors. Overset grid CFD schemes proceed by computing numerical solutions for each task and then updating boundary data across overlapping grids, generating the bulk of information transferred between the processors hosting the tasks. Effective task assignment schemes must therefore not only balance the computations but also reduce

*Department of Computer Science, Texas Tech University, Lubbock, TX 79409-3104 (nlb@cs.ttu.edu).

†Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA 94035-1000 (djomehri@nas.nasa.gov).

‡NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 (Rupak.Biswas@nasa.gov).

interprocessor communication costs.

Several partitioning schemes for load balancing exist [22], but most are static in nature and not suitable for dynamic reconfigurations. The task graph (TG) model proposed in this paper is used to represent a discrete step of the overset grid CFD simulation process. It is thus able to handle dynamic load balancing requirements by modifying the TG from one step to the next as necessary. The model also enables us to explore the feasibility of several allocation schemes so that the constraints inherent in the underlying applications are observed. The heuristics developed are based on two primitive assignments: smallest task first (STF) and largest task first (LTF). These assignments are subsequently enhanced by the systematic integration of the status of the processing units and the interprocessor communication costs. To evaluate and compare these proposed heuristics, a synthetic TG is generated where tasks are defined in terms of the number of grid points in predetermined overlapping grids. A TG derived from a realistic problem with eight million grid points is also used as a test case.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the OVERFLOW CFD code and cites some related work. Section 3 describes the TG representation of a discrete step in OVERFLOW, while Section 4 presents our proposed task assignment heuristics. Section 5 explains how synthetic TGs are generated for the purpose of evaluating the heuristics. Detailed performance results for synthetic and real TGs are presented and discussed in Section 6. Finally, Section 7 concludes the paper with a summary and some key observations.

2. Preliminaries. OVERFLOW [3], NASA's high-fidelity overset grid CFD production code, owes its popularity within the aerodynamics community due to its ability to handle complex configurations. These designs typically consist of multiple geometric components, where individual body-fitted grids can be constructed easily about each component. The grids are either attached to the aerodynamics configuration (near-body) or detached (off-body). The union of all near- and off-body grids covers the entire computational domain. In this work, we use a special version of OVERFLOW called OVERFLOW-D [16].

Both OVERFLOW and OVERFLOW-D use a Reynolds-averaged Navier-Stokes solver, augmented with a number of turbulence models. However, unlike OVERFLOW which is primarily meant for static grid systems, OVERFLOW-D is explicitly designed to simplify the modeling of components in relative motion (dynamic grid systems). At each time step, the flow equations are solved independently on each zone in a sequential manner. Overlapping boundary inter-grid data is updated from previous solutions prior to the start of the current time step using a Chimera interpolation technique [23]. OVERFLOW-D uses finite differences in space, with a variety of implicit/explicit time stepping.

Parallelization of OVERFLOW-D has been developed around its multi-block feature which offers a natural coarse-grained parallelism based on the message passing programming model. The MPI library is used to communicate the overlapping boundary data across processes. To facilitate parallel execution, a grouping strategy is used to assign each grid to an MPI process; however, parallel efficiency of the overset approach depends critically on how this grouping is performed. A number of simple and sophisticated grouping strategies for overset applications are discussed in [5].

The Chimera interpolation procedure [23] determines the proper connectivity of the individual grids. Adjacent grids are expected to have at least a one-cell (single fringe) overlap to ensure the continuity of the solutions; for higher-order accuracy and

to retain certain physical features in the solution, a double fringe overlap is sometimes used. A program named Domain Connectivity Function (DCF) [17] computes the inter-grid donor points that have to be supplied to other grids. The DCF procedure is incorporated into the OVERFLOW-D code and fully coupled with the flow solver. All boundary exchanges are conducted at the beginning of every time step based on the interpolatory updates from the previous time step. For dynamic grid systems, DCF has to be invoked at every time step to create new inter-grid boundary data.

In this work, a task graph (TG) is used to represent the interaction between the overset grids. Each grid is represented by a node in the TG, and a pair of overlapping grids is indicated by an edge between the corresponding nodes. Formally, a TG $G(V, E)$ consists of a set of vertices $V = \{v_i\}$, $i \geq 1$ to represent tasks, and a set of edges $E = \{e_i\}$, $i \geq 1$ to represent precedence constraints. If the execution time of each task is constant, calculating the job completion time is straightforward when assuming an unrestricted number of processing units. However, in a networked system, a task's execution time depends on the characteristics of the processor to which it is mapped while communication times depend on the latency and bandwidth of the interconnect. Thus, estimating and minimizing the total job completion time becomes an optimization problem that involves the proper scheduling of tasks to processors. An optimal non-preemptive schedule of independent tasks to be executed on a two-processor system is NP-complete [10]. However, linear and polynomial mapping times can be achieved if the structure of the TG is restricted; such is the case for the two-level directed acyclic graphs reported in [15]. To reduce complexity and make the procedure feasible for dynamically load balancing OVERFLOW-D TGs on large numbers of processors at the risk of obtaining sub-optimal results, novel scheduling heuristics are developed and presented in this paper.

A solution technique for series-parallel TGs is reported in [20, 21] as part of a software package called SHARPE. Other related work combines TGs and queueing theory; an analytical approach is presented in [1] based on the solution of synchronous queueing networks. Also, [24] reports the use of a hierarchical approach that combines Markov models and TGs. In [14], this combination is applied to the performance prediction of TGs executing in shared-memory multiprocessor environments. Stochastic Petri Nets (SPN) are also used to represent parallel programs. In [2], a set of translation rules maps language constructs into SPN-based segments that are then used for automatic translation of parallel programs. Simple SPN-based models can also capture precedence constraints and the restrictions imposed by assignment heuristics [12, 13]. The use of SPNs to represent TGs extends the analysis to obtain probability distributions of completion times as well as average execution times for different heuristics and computational configurations. Another advantage is that system scalability can be predicted as additional processing units become available.

3. Task Graph Representation. The use of a task graph (TG) to represent overset grid CFD simulations makes the performance prediction of several assignment heuristics possible. Task allocation heuristics attempt to minimize total job execution time based on criteria such as individual task run times and the volume of data exchanged.

Let $Z = \{Z_1, Z_2, \ldots, Z_m\}$ define a collection of m zones in the CFD problem. A large zone Z_i can be further subdivided into a fixed number of k_i subzones, such that $Z_i = \{Z_{i1}, Z_{i2}, \ldots, Z_{ik_i}\}$. Note that a partitioned Z_i consists of $k_i > 1$ non-overlapping sets of grid points [6]. We use the term PZ to refer to these partitioned zones. In the presence of PZs, the heuristics must also consider constraints such

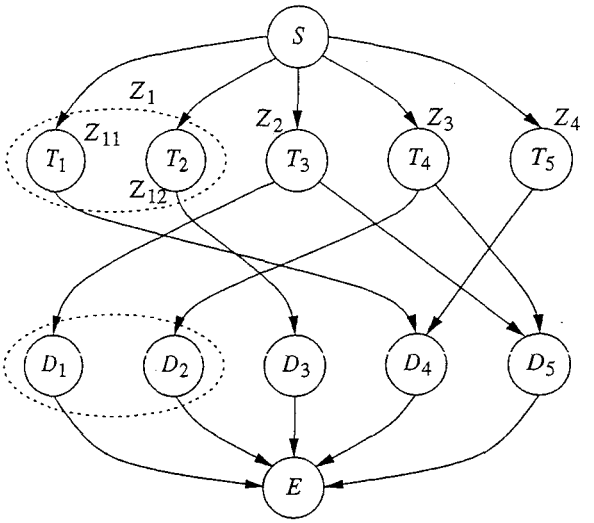

FIG. 3.1. *Task graph representation of a single iteration of an overset grid CFD application*

as pre-assignments to different processors to guarantee a parallel execution of the corresponding tasks. PZs are indicated by the dashed oval in Fig. 3.1 which illustrates a TG representation of a single iteration of the CFD problem. The set Z is clearly identified in the TG. It is convenient to map Z into a linear set T of q tasks T_i with no dependencies such that all tasks become identified with a single index. For example, the set $T = \{T_1, \{T_2, T_3, T_4\}, \{T_5, T_6, T_7, T_8\}, T_9, \ldots, T_q\}$ identifies a set Z with zones $Z_1 = T_1$, $Z_2 = \{T_2, T_3, T_4\}$, $Z_3 = \{T_5, T_6, T_7, T_8\}$, $Z_4 = T_9$, and so on, including $Z_m = T_q$. Note that, in this case, Z_2 and Z_3 are PZs containing three and four subzones, respectively. Dummy tasks D_i with zero execution times represent tasks receiving data needed for the next iteration of the computational process. An iteration starts and ends at nodes S and E, respectively. The arcs represent interactions between tasks; weights can be associated to these arcs to represent the volume of data transferred.

4. Task Assignment Heuristics. In this section, we describe our task assignment heuristics. To guarantee parallel execution, all tasks corresponding to PZs must be pre-assigned to different processors and coordinated to begin execution simultaneously. This is an important constraint enforced by each heuristic. Once the pre-assignment of PZs is made, the remaining tasks are allocated based on the assignment criteria. For q tasks and n processors, several tasks will end up on the same processor since $q \gg n$. All tasks are executed by the processors in the order in which they are assigned into the n groups: G_j, $j = 1, 2, \ldots, n$.

Let $E(G_j)$ denote the computation time of all the tasks in group G_j (i.e. assigned to processor P_j), and let X_i denote the computation time of task T_i. Then,

$$E(G_j) = \sum_{T_i \in G_j} X_i \tag{4.1}$$

and the execution time E of a single iteration is

$$E = \max_j \{E(G_j)\} \tag{4.2}$$

Assuming data generated by the tasks in G_j is routed in a serial fashion to other tasks, the transfer cost $C(G_j)$ of all tasks in G_j is

$$C(G_j) = \sum_{T_i \in G_j} \sum_k c_{ik} \tag{4.3}$$

where c_{ik} is the cost of sending data from task T_i to task T_k. The communication cost c_{ik} is assumed zero if both T_i and T_k are assigned to the same processor. Instead, if T_i and T_k are assigned to processors P_x and P_y, respectively, it is estimated [4] as

$$c_{ik} = (L_{xy} + v_{ik}/B_{xy}) a_{ix} a_{ky} \tag{4.4}$$

where L_{xy} and B_{xy} are the latency and bandwidth between P_x and P_y, v_{ik} is the volume of data generated by T_i with T_k as destination, and a_{ij} is a binary entry of an assignment matrix A^{qn}. The entry a_{ij} is set to one if T_i is assigned to P_j; otherwise, it is zero. An indicator function $I(x)$ that returns unity if the argument x is true is used to determine the values of a_{ij}. The specific form of $I(x)$ depends on the assignment heuristic. An independent model could also be used to obtain a better estimate of the communication costs.

By combining Eqs. 4.1 and 4.3, the execution time E_+ of a single iteration is obtained as

$$E_+ = \max_j \{E(G_j) + C(G_j)\} \tag{4.5}$$

Note that Eq. 4.5 enhances Eq. 4.2 by including the interprocessor communication overhead. Since this scheme separates computation and communication times, assignment heuristics can be developed based only on task computation times followed by estimated communication costs.

The following assumptions need to be highlighted at this point:

- Each CFD iteration is synchronized across processors and cannot commence execution until all data generated in the previous iteration is in place.
- Processors may be idle while communication takes place. They could also be idle as a result of the synchronization of iterations.
- The total application execution time ξ is scaled such that $\xi = \alpha E$. The term α is the number of iterations required to complete the entire CFD simulation.

Note that an upper bound of processor idle time IT can be obtained as

$$IT = E_+ - \min_j \{E(G_j) + C(G_j)\} \tag{4.6}$$

This is a global measure that could be useful as an objective function to be optimized if idle times for each processor were used during the assignment process.

Another metric that could be used as a measure of the effectiveness of an assignment is the load imbalance factor LIF:

$$LIF = \frac{\sum_j (E(G_j) + C(G_j))}{nE_+} \tag{4.7}$$

For heuristics that do not consider communication costs, the term $C(G_j)$ is dropped and E_+ is replaced with E in Eq. 4.7 when calculating LIF.

Let us now describe our proposed task assignment heuristics. The first two are our basic strategies and depend on whether the smallest or the largest tasks are assigned first. The next four enhance these basic techniques by incorporating the status of the processing units in terms of their minimum finish times or largest idle times. Finally, the last four further integrate the interprocessor communication costs. The overall relationship among the 10 heuristics is shown in Fig. 4.1. Note that the assignment matrix A^{qn} is updated each time a task is allocated to a processor using the indicator function $I(x)$ as specified below for each heuristic.

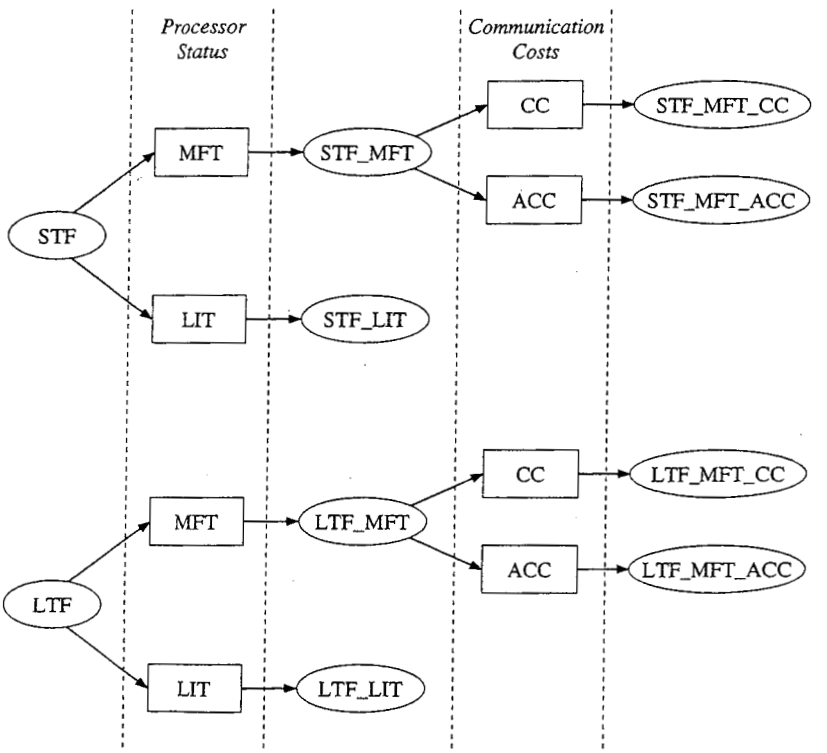

FIG. 4.1. *Overall relationship among our 10 proposed task assignment heuristics*

4.1. Smallest Task First (STF). In this scheme, the smallest unassigned task T_i is allocated to the next processor P_j selected from a list sorted in ascending order of their index. The assignment is determined such that

$$a_{ij} = I(X_i = \min_k \{X_k\}) \tag{4.8}$$

where X_k's are the execution times of all the unassigned tasks. In other words, tasks are assigned in ascending order of their execution time to processors in round-robin

fashion. All tasks corresponding to a PZ are assigned first; in this case, they are selected randomly within a single PZ set. If clustered processors are properly mapped into this sorted list, then most PZs are likely to be assigned to clustered processors and communication costs reduced accordingly.

4.2. Largest Task First (LTF). This heuristic is the inverse of STF and reported in [19]. An assignment is determined as

$$a_{ij} = I(X_i = \max_k\{X_k\}) \tag{4.9}$$

Here, processor P_j is again the next processor from a list sorted in ascending order by index; however, the tasks are now sorted in descending order of their computation times. As in STF, all tasks corresponding to a PZ are assigned first. Note that LTF will give the same assignment as STF because it is merely a maxsort versus minsort of the tasks.

4.3. STF with Minimum Finish Time (STF_MFT). The minimum accumulated time Ac_j at processor P_j is combined with the STF criteria. Thus, task T_i is assigned to processor P_j such that

$$a_{ij} = I(Ac_j = \min_i\{X_i\} + \min_j\{Ac_j\}) \tag{4.10}$$

In this scheme, the unassigned task T_i with the minimum computation time X_i is allocated to the processor that becomes free in the shortest time. Note that STF_MFT will result in the same assignment as STF since the latter, by design, automatically allocates the next task to the processor with the minimum finish time. This strategy is a variation of the heuristic reported in [18].

4.4. LTF with Minimum Finish Time (LTF_MFT). This scheme is a variation of Eq. 4.10 as the task with the maximum execution time is scheduled instead. Hence,

$$a_{ij} = I(Ac_j = \max_i\{X_i\} + \min_j\{Ac_j\}) \tag{4.11}$$

This task assignment heuristic is similar to the bin-packing strategy described in [6]. Note that LTF-based assignment heuristics should generally perform significantly better than the corresponding STF-based strategies because the largest tasks are allocated first.

4.5. STF with Largest Idle Time (STF_LIT). This scheme is another alternative that combines STF and the largest idle time of a processor. It is similar to STF_MFT except that an upper bound of the idle time is determined by the maximum accumulated time. Let h_j denote the current idle time of processor P_j with respect to $\max_k\{Ac_k\}$. Then,

$$h_j = \max_k\{Ac_k\} - Ac_j \tag{4.12}$$

Thus, initially all h_j evaluate to zero and the first n tasks chosen randomly (assuming they can execute in parallel) are assigned to the n processors. Thereafter, h_j is selected

begin procedure STF_LIT
1. Compute h_j using Eq. 4.12 and sort in descending order
2. Select next task T_i such that its X_i is minimum
3. Assign T_i to processor P_j such that its h_j is maximum
4. Update a_{ij} accordingly
5. If $(\Delta_{ij} = \max_j\{h_j\} - \min_i\{X_i\} < 0)$, repeat from step 1
6. Else repeat from step 2

end procedure STF_LIT

FIG. 4.2. *Assignment procedure for the STF_LIT heuristic*

from a descending order list and X_i from an ascending order list. The task T_i with the smallest X_i is assigned to the processor with the largest h_j. The assignment of these remaining tasks is summarized in Fig. 4.2. Note that every time the h_i list is modified, it is reordered again with an added sorting cost of $O(n \log n)$. The h_j values change depending on $\Delta_{ij} = \max_j\{h_j\} - \min_i\{X_i\}$.

4.6. LTF with Largest Idle Time (LTF_LIT). This scheme is a variation of STF_LIT except that all computation times are now sorted in descending order.

4.7. STF_MFT with Communication Costs (STF_MFT_CC). We now incorporate the interprocessor communication costs into the task assignment heuristics. However, STF_LIT and LTF_LIT are no longer considered because their performance is similar to STF_MFT and LTF_MFT, respectively. We first ignore the fact that some destination tasks may not yet be allocated to processors and therefore the communication cost must be estimated. The STF_MFT_CC strategy assigns non-PZ tasks with minimum computation times and communication costs to processors with the current minimum finish time. The assignment matrix is created as

$$a_{ij} = I(Ac_j = \min_i\{X_i + C_i\} + \min_j\{Ac_j\}) \tag{4.13}$$

Note that the only modification to the STF_MFT indicator function in Eq. 4.10 is that a task is selected such that the sum of its computation and communication times is minimum. This value is then added to the accumulated time of the selected processor. The communication time for task T_i is obtained as $C_i = \sum_k c_{ik}$, where the index k identifies all destination tasks. A drawback of this scheme is that all destination tasks are required to be already allocated. If this is not the case, the communication cost is estimated by assuming constant latency L and bandwidth B, and then using Eq. 4.4. Our implementation steps are shown in Fig. 4.3. The adjustments in step 3 are required because no communication costs are incurred if two interacting tasks are assigned to the same processor.

4.8. LTF_MFT with Communication Costs (LTF_MFT_CC). Task assignments under this scheme are similar to those represented by Eq. 4.13 except that they are now conducted with respect to the maximum value of $(X_i + C_i)$:

$$a_{ij} = I(Ac_j = \max_i\{X_i + C_i\} + \min_j\{Ac_j\}) \tag{4.14}$$

begin procedure STF_MFT_CC
1. Sort tasks T_i in ascending order in terms of $(X_i + C_i)$
2. For each processor P_j, set $Ac_j = 0$
3. For each task T_i allocated to processor P_j
   $Ac_j = Ac_j + X_i + C_i$
4. For each task T_i allocated to processor P_j
   4.1. If a destination task T_k is assigned to the same processor P_j, do
   $Ac_j = Ac_j - c_{ik}$
   4.2. If a predecessor task T_h is assigned to the same processor P_j, do
   $Ac_j = Ac_j - c_{hi}$

end procedure STF_MFT_CC

FIG. 4.3. *Assignment procedure for the STF_MFT_CC heuristic*

4.9. STF_MFT with Actual Communication Costs (STF_MFT_ACC). In this scheme, tasks are allocated to processors according to Eq. 4.10 and communication costs are calculated only after predecessor and destination tasks are assigned. Network latency and bandwidth depend on where the interacting tasks are actually allocated. The addition of c_{ik} whenever possible updates the processor accumulated time that integrates communication costs at least partially. An outline of the implementation algorithm is shown in Fig. 4.4.

begin procedure STF_MFT_ACC
1. Sort tasks T_i in ascending order in terms of X_i
2. For each processor P_j, set $Ac_j = 0$
3. For each task T_i in sorted list
   3.1. Assign task T_i to processor P_j with minimum Ac_j
   $Ac_j = Ac_j + X_i$
   3.2. For each task $T_r \in R(T_i)$ assigned to processor $P_k \neq P_j$, do
   $Ac_j = Ac_j + c_{ir}$
   3.3. For each task $T_d \in D(T_i)$ assigned to processor $P_m \neq P_j$, do
   $Ac_m = Ac_m + c_{di}$

end procedure STF_MFT_ACC

FIG. 4.4. *Assignment procedure for the STF_MFT_ACC heuristic*

As task T_i is assigned, the set of tasks $R(T_i)$ receiving data from T_i but assigned to a different processor is updated. The accumulated time of the processor hosting T_i is also adjusted with the communication costs for all tasks in $R(T_i)$. Likewise, the set of predecessor tasks $D(T_i)$ donating data to T_i but assigned to a different processor is updated. The accumulated time of all processors hosting tasks in $D(T_i)$ is also updated. To illustrate how STF_MFT_ACC operates, consider the simple TG in Fig. 4.5 and the execution steps in Table 4.1 when assigning it to a two-processor system.

4.10. LTF_MFT with Actual Communication Costs (LTF_MFT_ACC). The procedure given in Fig. 4.4 for STF_MFT_ACC apply here except that the allocation follows Eq. 4.11 for which tasks are sorted in descending order of their computation times. Table 4.2 shows the execution steps when assigning the TG in Fig. 4.5 to a two processors. Notice that the workload is balanced much better than in Table 4.1.

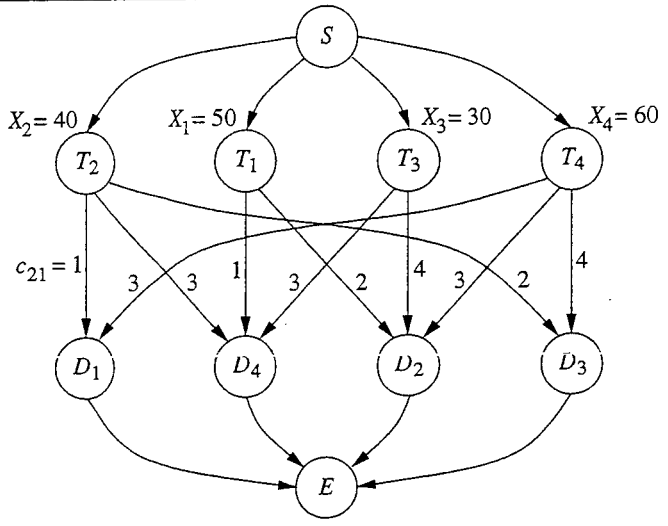

FIG. 4.5. *Task graph of a small overset grid CFD application*

TABLE 4.1
Execution trace of STF_MFT_ACC for the TG in Fig. 4.5

1.	Sorted tasks T_3, T_2, T_1, T_4	3.1.	$T_1 \rightarrow P_1$
2.	$Ac_1 = Ac_2 = 0$		$Ac_1 = 34 + 50 = 84$
3.1.	$T_3 \rightarrow P_1$	3.2.	$R(T_1) = \{T_2\}$
	$Ac_1 = 0 + 30 = 30$		$Ac_1 = 84 + 2 = 86$
3.2.	$R(T_3) = \emptyset$	3.3.	$D(T_1) = \{T_2\}$
3.3.	$D(T_3) = \emptyset$		$Ac_2 = 42 + 1 = 43$
3.1.	$T_2 \rightarrow P_2$	3.1.	$T_4 \rightarrow P_2$
	$Ac_2 = 0 + 40 = 40$		$Ac_2 = 43 + 60 = 103$
3.2.	$R(T_2) = \{T_3\}$	3.2.	$R(T_4) = \{T_1, T_3\}$
	$Ac_2 = 40 + 2 = 42$		$Ac_2 = 103 + 3 + 4 = 110$
3.3.	$D(T_2) = \{T_3\}$	3.3.	$D(T_4) = \{T_1, T_3\}$
	$Ac_1 = 30 + 4 = 34$		$Ac_1 = 86 + 1 + 3 = 90$

TABLE 4.2
Execution trace of LTF_MFT_ACC for the TG in Fig. 4.5

1.	Sorted tasks T_4, T_1, T_2, T_3	3.1.	$T_2 \rightarrow P_2$
2.	$Ac_1 = Ac_2 = 0$		$Ac_2 = 51 + 40 = 91$
3.1.	$T_4 \rightarrow P_1$	3.2.	$R(T_2) = \{T_4\}$
	$Ac_1 = 0 + 60 = 60$		$Ac_2 = 91 + 3 = 94$
3.2.	$R(T_4) = \emptyset$	3.3.	$D(T_2) = \{T_4\}$
3.3.	$D(T_4) = \emptyset$		$Ac_1 = 63 + 3 = 66$
3.1.	$T_1 \rightarrow P_2$	3.1.	$T_3 \rightarrow P_1$
	$Ac_2 = 0 + 50 = 50$		$Ac_1 = 66 + 30 = 96$
3.2.	$R(T_1) = \{T_4\}$	3.2.	$R(T_3) = \{T_2, T_4\}$
	$Ac_2 = 50 + 1 = 51$		$Ac_1 = 96 + 4 + 3 = 103$
3.3.	$D(T_1) = \{T_4\}$	3.3.	$D(T_3) = \{T_2, T_4\}$
	$Ac_1 = 60 + 3 = 63$		$Ac_2 = 94 + 2 + 4 = 100$

5. Synthetic Task Graph Generation. All of our proposed heuristics can be evaluated with synthetic TGs. Given the total number of grid points ngp and the number of tasks q, the synthetic TG generation process consists of three steps. The first step is specifying the number of grid points for each zone Z_i. The procedure GEN_GRIDS outlined in Fig. 5.1 accomplishes this goal. Basically, each Z_i is initially allocated a random number of grid points less than $\lfloor ngp/q \rfloor$. If the total number of grid points for all q zones is less than ngp, the remaining grid points are randomly added to one of the zones.

```
begin procedure GEN_GRIDS
    sum = 0
    for (i = 1 to i = q)
        Generate a random number x ∈ {1, ngp/q}
        Allocate x grid points to Z_i
        sum = sum + x
    end for
    if (sum < ngp)
        Generate a random index i ∈ {1, q}
        Allocate (ngp − sum) additional grid points to Z_i
    end if
end procedure GEN_GRIDS
```

FIG. 5.1. *Procedure for generating zones containing different numbers of grid points*

A second procedure called GEN_TG, outlined in Fig. 5.2, generates the topology of the TG with tasks defined by zones generated by GEN_GRIDS. The number of overlapping zones defines a window of size $w = r \times q \times o$ where r is a random number between 0 and 1, and o is a user-supplied parameter that specifies the maximum percentage of q zones that can overlap. Each zone Z_i may overlap with $w/2$ zones below and above its index i.

```
begin procedure GEN_TG
    for (i = 1 to i = q)
        Generate a random number r ∈ {0,1}
        w = int (o × r × q)
        count = 0
        for (j = 1 to j = w/2)
            l = i − j
            if (l < 1) l = l + q
            if (count + + = w) break
            Overlap zone i with zone l
            u = i + j
            if (u > q) u = u − q
            Overlap zone i with zone u
        end for
    end for
end procedure GEN_TG
```

FIG. 5.2. *Procedure for generating the topology of a synthetic TG*

Finally, a third procedure called COMM_VOL generates the volume of data ex-

```
begin procedure COMM_VOL
    for (i = 1 to i = q)
        m = number of overlapping zones for Z_i
        for (j = 1 to j = m)
            Obtain index k of j-th overlapping zone
            c_ik = rc × X_k
        end for
    end for
end procedure COMM_VOL
```

FIG. 5.3. *Procedure for generating communication volume between overlapping zones*

changed between interacting tasks in terms of the number of overlapping grid points. The outline of this procedure is shown in Fig. 5.3. This program reads a file (generated by GEN_GRIDS) containing the number of grid points X_i for each task and a file (generated by GEN_TG) containing the indices of overlapping tasks. The communication volume between task T_i and T_j is set as $c_{ij} = rc \times X_j$ where rc is the fraction of grid points that are in the overlap region.

6. Performance Results. A simple interface called EValuate Assignment Heuristic (EVAH) has been implemented to compare and contrast the various heuristics. We first generated a synthetic TG containing 128 zones and 16 million grid points. The other parameters are set as follows: computation time per grid point $xpg = 15$ μsec, fraction of grid points in overlap region $rc = 0.5$, network latency $L = 13$ μsec, and network bandwidth $B = 37.3$ Mbytes/sec. Latency and bandwidth are required to estimate the cost of data exchange between overlapping zones. The communication volume is calculated in terms of the number of overlapping grid points multiplied by a factor equal to the number of bytes per grid point required to exchange data. In our case, this factor is assumed to be 200 bytes. All evaluations are performed assuming the number of processors to be between 2 and 128.

Two performance metrics are reported: load imbalance factor (LIF) and speedup (S). LIF is calculated using Eq. 4.7, while S is computed as $S = (ngp \times xpg)/E_+$, where E_+ is given by Eq. 4.5. Fig. 6.1 shows the LIF for all the heuristics based on STF, while Fig. 6.2 shows results for those based on LTF. Results for STF_LIT and LTF_LIT are not presented because they are almost identical to those for STF_MFT and LTF_MFT, respectively. Both figures show results for the baseline STF and LTF heuristics for sake of comparison. While the general trends are similar, improvements are more significant for the LTF-based heuristics. The best overall results are obtained with LTF_MFT_CC. This is expected for homogeneous processing systems. For heterogeneous environments, as in computational grids, the LTF_MFT_ACC heuristic should do significantly better.

Performance results for speedup S for the STF- and LTF-based heuristics are shown in Figs. 6.3 and 6.4. Again, the best results are achieved by LTF_MFT_CC. If the underlying system architecture possesses uniform interprocessor communication characteristics, the obvious choice is LTF_MFT_CC. Otherwise, one should use LTF_MFT_ACC which allows only actual communication costs to influence the task assignment scheme.

Notice that the performance metrics are identical for all heuristics when the number of processors n is 1 or 128. When $n = 1$, we are simulating sequential execution of the TG; hence, task assignment does not have any effect. When $n = 128$, each

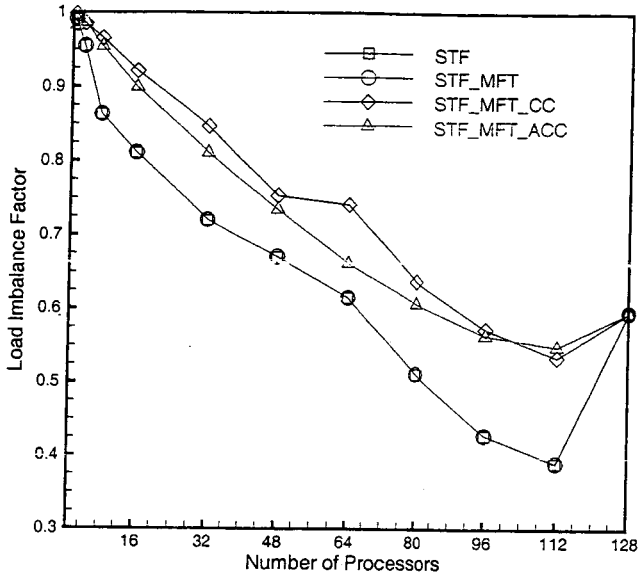

FIG. 6.1. *Load imbalance factors for STF-based heuristics on synthetic TG*

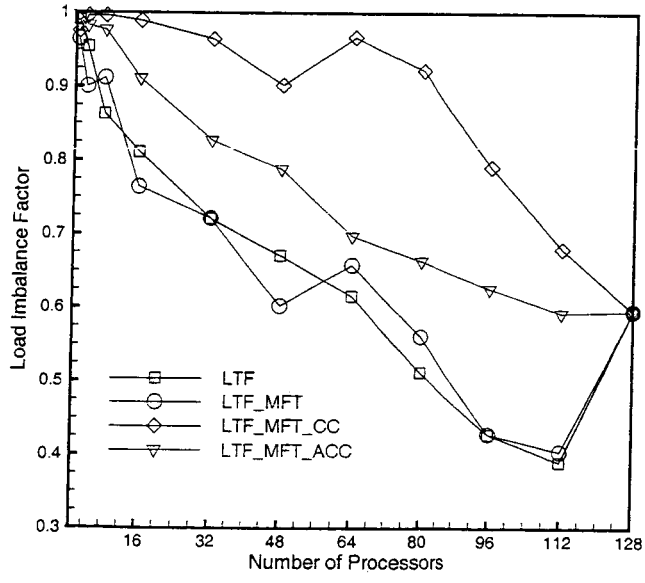

FIG. 6.2. *Load imbalance factors for LTF-based heuristics on synthetic TG*

processor executes exactly one zone since our synthetic TG has 128 zones. Thus, all assignment heuristics return the same result. However, the actual trend between 2 and 127 processors depends on the assignment scheme and the TG.

We now present results obtained using a real test case of eight million grid points

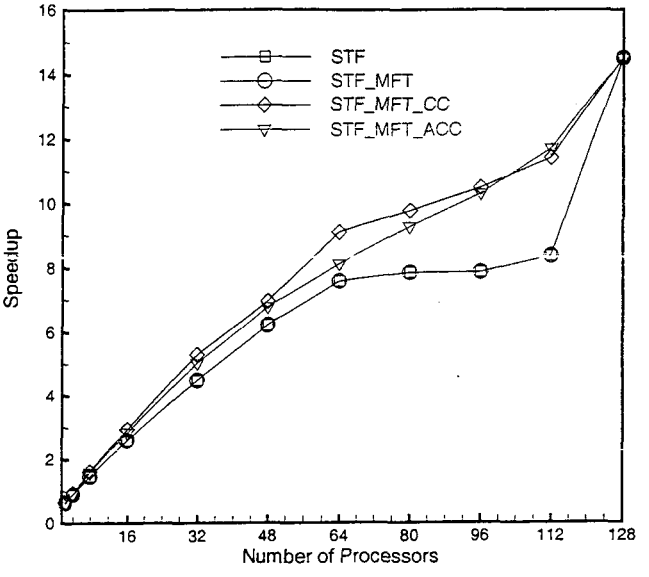

FIG. 6.3. *Speedups for STF-based heuristics on synthetic TG*

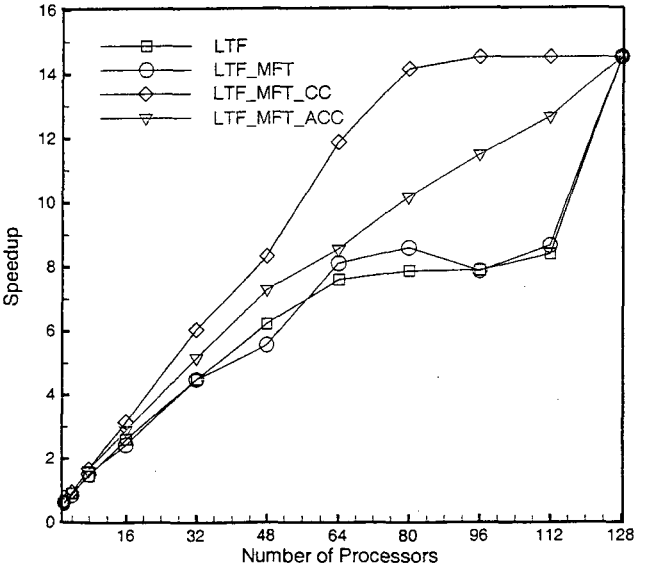

FIG. 6.4. *Speedups for LTF-based heuristics on synthetic TG*

distributed among 41 zones. The plots in Figs. 6.5 and 6.6 present detailed comparisons of computation and communication times per processor as predicted by the LTF_MFT_ACC assignment heuristic and those actually computed by OVERFLOW-D using LTF_MFT_ACC on an SGI Origin3000, for 16 and 32 processors. For the

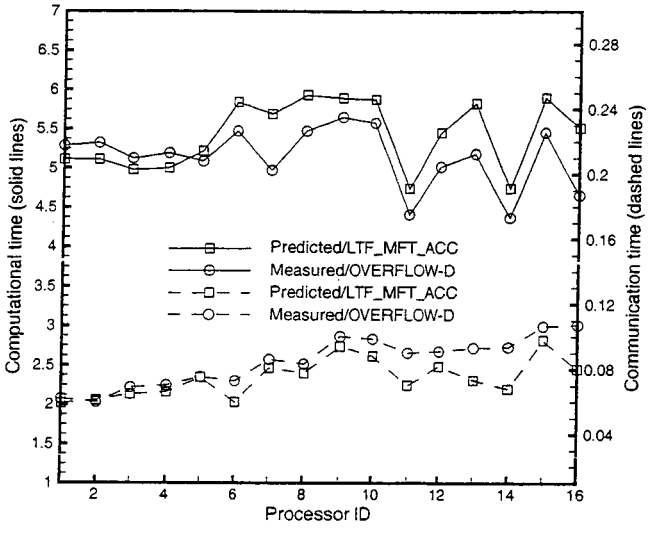

FIG. 6.5. *Computation and communication times per processor for 16 processors on TG obtained from real test case*

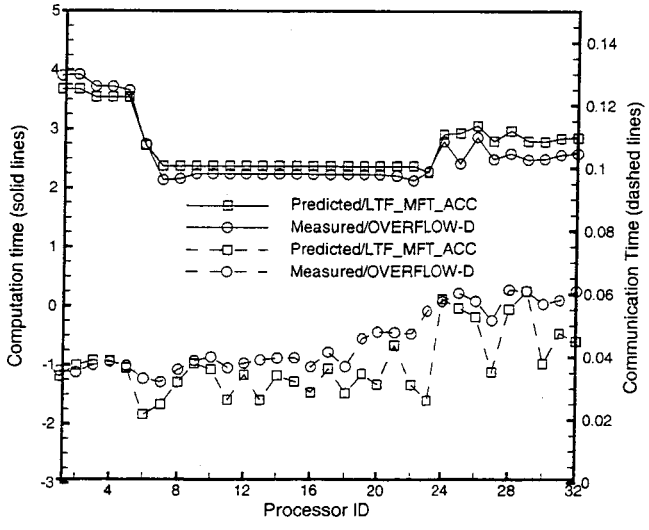

FIG. 6.6. *Computation and communication times per processor for 32 processors on TG obtained from real test case*

sake of clarity, plots of computation and communication times are shown at different scales indicated by the left and right vertical axes, respectively.

The predicted computation times closely match the measured data for both $n = 16$ and $n = 32$. The predicted communication times are slightly off, especially for certain processors in the $n = 32$ case. These mismatches are primarily due

to the assumptions in the task assignment heuristic. For example, LTF_MFT_ACC assumes certain values for network latency and bandwidth that are then used to calculate interprocessor communication costs. These parameters are estimated from the 16-processor run, which explains the bigger discrepancy for $n = 32$. However, these parameters are not constants but change dynamically at run time depending on message size and interconnect topology, features that are not modeled by our assignment heuristics. The computation time per grid point, on the other hand, is relatively accurate and uniform across processors for a tightly coupled parallel platform such as the Origin3000. As a result, the predicted computation times match better with actual data.

For applications running on small numbers of processors, such as the ones shown in Figs. 6.5 and 6.6, the communication cost is only a small fraction (less than 2%) of the computation time. The predicted total execution time is therefore hardly affected by the error in estimating the communication time, and matches well with the measured data. Also, the larger spread in computation times for $n = 32$ reflects the general difficulty with load balancing as the number of zones approaches the number of processors. Results using larger test cases and more processors can be found in [5].

TABLE 6.1
Comparison of various task assignment heuristics for the real TG on 16 processors

Heuristic	E (Eq. 4.2)	E_+ (Eq. 4.5)	IT (Eq. 4.6)	LIF (Eq.4.7)
STF	7.464	7.553	3.687	0.729
STF_MFT	7.464	7.553	3.687	0.729
STF_LIT	7.464	7.553	3.687	0.729
STF_MFT_CC	7.464	7.550	3.684	0.729
STF_MFT_ACC	7.464	7.553	3.687	0.729
LTF	7.464	7.559	3.696	0.728
LTF_MFT	5.933	6.006	1.190	0.917
LTF_LIT	5.933	6.013	1.205	0.916
LTF_MFT_CC	5.933	6.015	1.205	0.915
LTF_MFT_ACC	5.933	6.011	1.193	0.916

Finally, Table 6.1 lists results obtained with all the task assignment heuristics using the real TG on 16 processors of an Origin3000. Observe that except for the basic LTF heuristic, all the other LTF-based strategies are significantly better than the STF-based schemes. This is expected because the largest tasks are allocated first in the LTF heuristics. Incorporating the communication cost model has very little effect in this case since the communication times are negligible compared to the total execution times.

7. Discussion and Conclusions. In this paper, we first presented a task graph (TG) model to represent a single step of multi-block overset grid computational fluid dynamics (CFD) applications. The nodes of the TG correspond to the individual grids, while an edge between two nodes indicate overlapping grids. We then described a set of task assignment heuristics tailored to meet the load balance requirements of this class of CFD problems on high performance parallel and distributed systems. The heuristics were derived by first considering two basic criteria to assign tasks to processors: smallest task first (STF) and largest task first (LTF). These heuristics were then systematically enhanced by integrating the status of the processing units in

terms of their minimum finish times or largest idle times. Finally, the heuristics were modified in a way that interprocessor communication costs would reflect the type of network being used. A synthetic TG containing 128 grids and 16 million grid points was used to study and compare the behavior of all the assignment schemes. A TG obtained from a real test case with eight million grid points and 41 grids was also analyzed and compared with measured data.

The work reported in this paper is targeted to CFD users and intended for eventual development of dynamic assignment schemes with minimum overhead. Data exchanges are currently assumed to have uniform cost; however, a realistic prediction of communication must take into account the variety and location of the computing resources. Future enhancements will include a user-friendly iterative procedure to enable scientists and engineers achieve optimal scalability across available resources. While the emphasis of this work is performance prediction, suitable partitioning schemes such as MeTiS [11] can be used to provide an initial grid partition.

REFERENCES

[1] F. Baccelli and A. Liu. On the execution of parallel programs on multiprocessor systems — a queueing theory approach. *Journal of the ACM*, 37(2):373–414, April 1990.

[2] G. Balbo, S. Donatelli, and G. Franceschinis. Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171–187, July 1992.

[3] P. G. Buning, D. C. Jespersen, T. H. Pulliam, W. M. Chan, J. P. Slotnick amd S. E. Krist, and K. J. Renze. Overflow user's manual, version 1.8g. Technical report, NASA Langley Research Center, Hampton, VA, 1999.

[4] G. Chiola and G. Ciaccio. Lightweight messaging systems. In R. Buyya, editor, *High Performance Cluster Computing: Architectures and Systems*, pages 246–268. Prentice Hall, Upper Saddle River, NJ, 1999.

[5] M. J. Djomehri, R. Biswas, and N. Lopez-Benitez. Load balancing strategies for multi-block overset grid applications. In *Proc. 18th International Conference on Computers and Their Applications*, pages 373–378, Honolulu, HI, March 2003.

[6] M. J. Djomehri, R. Biswas, R. F. Van der Wijngaart, and M. Yarrow. Parallel and distributed computational fluid dynamics: Experimental results and challenges. In *Proc. 7th International Conference on High Performance Computing*, volume LNCS 1970, pages 183–193, Bangalore, India, December 2000.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[8] I. Foster and C. Kesselman. The Globus toolkit. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278. Morgan Kaufmann, San Francisco, CA, 1999.

[9] W. E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. In *Proc. 8th International Symposium on High Performance Distributed Computing*, pages 197–204, Redondo Beach, CA, August 1999.

[10] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, NY, 1972.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, Dept. of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[12] N. Lopez-Benitez. Petri-net based performance evaluation of distributed homogeneous task systems. *IEEE Transactions on Reliability*, 49(2):188–198, June 2000.

[13] N. Lopez-Benitez and J.-Y. Hyon. Simulation of task graph systems in heterogeneous computing environments. In *Proc. 8th Heterogeneous Computing Workshop*, pages 112–124, San Juan, Puerto Rico, April 1999.

[14] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[15] P. Markenscoff and Y. Y. Li. Scheduling a computational DAG on a parallel system with

communication delays and replication of node execution. In *Proc. 7th International Parallel Processing Symposium*, pages 113–117, Newport Beach, CA, April 1993.

[16] R. Meakin. On adaptive refinement and overset structured grids. In *Proc. 13th AIAA Computational Fluid Dynamics Conference*, Paper number 97-1858, Snowmass, CO, June 1997.

[17] R. Meakin and A. M. Wissink. Unsteady aerodynamic simulation of static and moving bodies using scalable computers. In *Proc. 14th AIAA Computational Fluid Dynamics Conference*, Paper number 99-3302, Norfolk, VA, 1999.

[18] D. A. Menasce, S. C. S. Porto, and S. K. Tripathi. Static heuristic processor assignment in heterogeneous multiprocessors. *International Journal of High Speed Computing*, 6(1):115–137, March 1995.

[19] D. A. Menasce, D. Saha, S. C. S. Porto, V. Almeida, and S. K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, July 1995.

[20] R. A. Sahner and K. S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engineering*, 13(10):1105–1114, October 1987.

[21] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems*. Kluwer, Boston, MA, 1996.

[22] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editor, *CRPC Parallel Computing Handbook*. Morgan Kaufmann, San Francisco, CA, 2000.

[23] J. L. Steger, F. C. Dougherty, and J. A. Benek. A Chimera grid scheme. In K. N. Ghia and U. Ghia, editors, *ASME FED-5*. 1983.

[24] A. Thomasian and P. F. Bay. Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers*, 35(12):1045–1054, December 1986.