Proceedings of Supercomputing '96

Application-Level Scheduling on Distributed Heterogeneous Networks * (Technical Paper)

Fran Berman and Rich Wolski†
Silvia Figueira, Jennifer Schopf, Gary Shao
Department of Computer Science and Engineering 0114
University of California, San Diego
La Jolla, Calif. 92093 ‡

Abstract

Heterogeneous networks are increasingly being used as platforms for resource-intensive distributed parallel applications. A critical contributor to the performance of such applications is the scheduling of constituent application tasks on the network. Since often the distributed resources cannot be brought under the control of a single global scheduler, the application must be scheduled by the user. To obtain the best performance, the user must take into account both application-specific and dynamic system information in developing a schedule which meets his or her performance criteria.

In this paper, we define a set of principles underlying **application-level scheduling** and describe our work-in-progress building **AppLeS** (application-level scheduling) agents. We illustrate the application-level scheduling approach with a detailed description and results for a distributed 2D Jacobi application on two production heterogeneous platforms.

1 Introduction

Fast networks have made it possible to coordinate distributed CPU, memory, and storage resources to provide the potential for application performance superior to that achievable from any single system [1]. Parallel applications targeted to such systems are typically resource-intensive, i.e. they require more resources than are available at a single site [16]. Critical resources may include large aggregated and distributed memory, fixed data sources,

*The authors were supported in part by NSF grants ASC-9301788, ASC-9308900, and a scholarship from CAPES and UFRJ (Brazil).

†Presenting author.

‡Email addresses of the authors are {berman, rich, silvia, jenny, gshao }@cs.ucsd.edu. Fran Berman's phone is 619-5346195 and fax is 619-5347029.

local temporary storage, and computational cycles. Performance is defined by the user, and may mean different things for different applications, however achieving it requires the efficient use of all relevant resources.

Despite the performance potential that distributed systems offer for resource-intensive parallel applications, actually achieving the user's performance goals can be difficult. One of the most fundamental problems that must be solved to realize good performance is the determination of an efficient schedule. Effective scheduling by the application developer or end-user involves the integration of application-specific and system-specific information, and is dependent on the dynamic interactions between an application and the relevant system(s).

Currently, the performance-seeking end-user must develop schedules for distributed heterogeneous applications off-line, using intuition to predict how the application will perform at the time it will execute. The users or application developers must select a configuration of resources based on load and availability, evaluate the potential performance of their application on such configurations (based on their own performance criteria), and interact with the relevant resource management systems in order to implement the application. At the same time, other users (running their own applications) draw from the same set of resources, each seeking to achieve his or her own performance goals. When multiple users contend for resources, only a fraction of the resource performance can be delivered to each.

In this paper, we describe an application-specific approach to scheduling individual parallel applications on production heterogeneous systems. We are developing software to facilitate and improve upon the scheduling activities of the user. Our goal is to develop scheduling agents that perform this task for the user at machine speeds and with more comprehensive information. We term these agents **AppLeS** – **Application-Level Schedulers**. Each application will have its own AppLeS to determine a performance-efficient schedule, and to implement that schedule with respect to the appropriate resource management systems.

Note that AppLeS is not a resource management system; rather, it interacts with systems such as Globus [3, 11], Legion [12, 17], or PVM [9, 20] to perform that function. As such, AppLeS is an **application-management system** which manages the scheduling of the application for the benefit of the end-user.

In the next subsection, we describe our approach for AppLeS.

1.1 Scheduling from the Perspective of the Application

Application-level scheduling is based on four underlying principles:

- **Application- and system-specific information is needed for good schedules.** Users determine good schedules for their applications based on their perception of system capabilities, and their knowledge of the structure and requirements of their application. The frequency of communication and computation, the amount of memory required, the number, type, and size of application data structures are matched with the granularity of the computational platforms, network speed and bandwidth, and other system attributes to develop a performance-efficient schedule.

- **Dynamic information is necessary to determine system state.**
  Users base candidate schedules on knowledge of which machines are available and which are heavily or lightly loaded. This load varies over time and with usage of system resources. If a choice of networks or computational platforms is available, the user will combine his/her knowledge of how the application will use the system with the current or predicted load on its resources.

- **Good schedules involve some prediction of application and system performance**.
  Prediction provides the basis for most scheduling. The user predicts how their application will execute on the system and uses this prediction to choose a performance-efficient schedule. Such predictions are difficult to make accurately since the system varies over time due to contention, and application performance may be dependent on both data and system load. However, simplifying the model of the system or application excessively to make the prediction task easier is not always fruitful. Optima for a simplified model may not correlate with optimal behavior in practice. In particular, application and system models must be sufficiently complex to expose real phenomena.

- **All resources can be evaluated strictly in terms of the performance they deliver to the application**.
  Notice that, from the perspective of the application (or user), each resource is judged ultimately on how much it benefits the application's execution. Users define different criteria for performance (speed, cost, etc.), but the decision about which resources to use, and when to use them, is based on how they will perform (in terms of the specific criteria) when executing the user's application.

The AppLeS approach is to use parameterizable application- and system-specific models to predict application performance using a given set of resources. Using these models in conjunction with forecasts of expected resource load, an AppLeS agent can select a resource set and an application schedule by evaluating various candidate mappings. The mapping that generates the best expected performance is chosen and implemented on the target resource management system(s).

Note that a fundamental difference between the AppLeS approach and system-oriented schedulers is that for AppLeS, **everything about the system is experienced from the point of view of the application**. If the candidate resources for the application are lightly loaded, then the system appears lightly loaded to the application regardless of the load on other resources. If the candidate resources are heavily loaded, then the system appears heavily loaded.

In the next section, we utilize the application-level scheduling approach to develop an efficient schedule for a distributed Jacobi data-parallel code. The example serves as a "proof of concept" for the principles underlying the AppLeS approach, and serves to illuminate the components required for general application-oriented scheduling agents. After discussing the Jacobi example in detail, we will describe our current efforts to build general AppLeS agents for scheduling in Section 4.

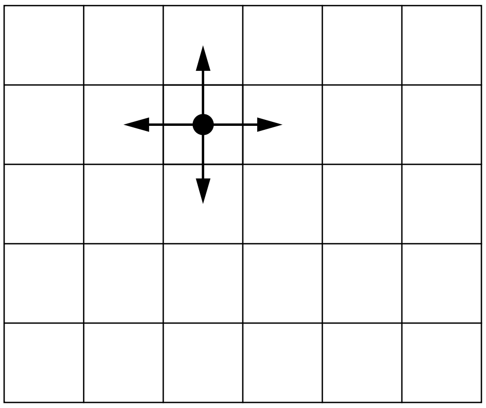

Figure 1: Five-point Jacobi Computation

2 Application-Level Scheduling of Jacobi2D

In this section, we illustrate and motivate our approach using a simple application. We discuss the development of an application-level schedule for a distributed 2D Jacobi application in detail and present performance data.

Consider the problem of executing a distributed data-parallel two dimensional Jacobi iterative solver (**Jacobi2D**) using a heterogeneous network of machines. The Jacobi method is commonly used to solve the finite-difference approximation to Poisson's equation [15] which arises in many heat flow, electrostatic, and gravitational problems. Variable coefficients are represented as elements of a two-dimensional grid. At each iteration, the new value of each grid element is defined to be the average of its four nearest neighbors during the previous iteration (see Figure 1).

Typically, the Jacobi computation is parallelized by partitioning the grid into rectangular regions, and then assigning each region to a different processor. This decomposition strategy is favorable because a processor need only obtain the border elements for its region during each iteration. The amount of computational work scales as the area of each region, whereas the amount of delay due to communication scales as the perimeter. A small number of big regions will yield good processor efficiencies, but may sacrifice parallelism. Conversely, a large number of small regions may incur large communication overhead. In our example, the user wishes to identify a partitioning that yields the lowest possible execution time. Solving the partitioning problem optimally is NP-complete, so it is necessary for the user to employ heuristics to arrive at a "good" solution.

2.1 Deriving Partitions that Optimize Resource Performance

The version of Jacobi2D we use in this example is written in a data-parallel SPMD style using KeLP [6, 5]. The KeLP system provides high-level abstractions, in the form of C++ objects, that support runtime data decomposition. In addition, the details associated with message passing in distributed-memory computing environments are buried in the abstrac-

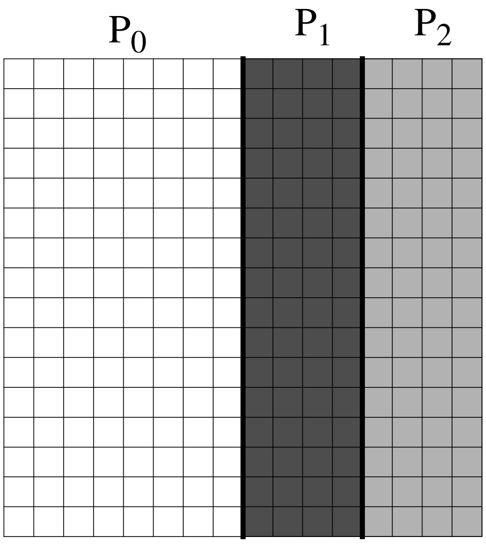

Figure 2: Strip data partitioning for three processors where processor P_0 is twice as fast as processor P_1 or P_2.

tions making the code portable and easy to maintain.

An ideal partitioning will assign regions of the Jacobi grid to processors such that the area of each region matches the performance capability of the processor to which it is assigned. Faster processors should compute over larger regions than slower ones. In particular, computational time is optimized when the ratio of each rectangular area of the grid to the total grid area most closely matches the ratio of the power of the processor to which the rectangular area is allocated to the total processing power available.

However, it is not simply a processor's computational time that defines its performance capability for Jacobi2D. The performance capability of each processor depends on how fast each processor can locally compute an element of the Jacobi matrix, and how quickly each processor can communicate its border elements with its neighboring processors. These two factors most dramatically affect execution time of this application.

To derive partitions that balance resource performance, we formulate the partitioning problem as an analytical model. Let

$$\begin{aligned} T_i &= \textit{time for processor } i \textit{ to compute region } i \\ A_i &= \textit{the area of region } i \\ P_i &= \textit{the time required for processor } i \textit{ to compute a single point locally} \\ C_i &= \textit{the time for processor } i \textit{ to send and receive its borders} \end{aligned}$$

for i in I regions and processors. The time each processor spends computing and communicating during a single iteration of Jacobi2D can then be represented as

$$T_i = A_i * P_i + C_i$$

This equation predicts the execution time (including the time spent communicating) for each processor. If all partitions are scheduled simultaneously, then the execution time for a single iteration will be equal to the maximum value of T_i. We can balance the time each processor spends computing and communicating by setting all T_i equal and solving the resulting system of equations for A_i. For a grid with N rows and M columns, let

$$T_1 = T_2 = T_3 = \ldots = T_I \tag{1}$$
$$\sum_{i=1}^{I} A_i = N * M$$

We restrict the legal partitions to those which only consider a single dimension (i.e. strip partitions, shown in Figure 2), so that C_i does not depend on A_i. For this type of partitioning, the system of equations (1) is linear and can be solved quickly by conventional methods.

For a strip partitioning, we let

$$C_i = \begin{cases} Recv(i-1,i) + Recv(i+1,i) + Send(i,i-1) + Send(i,i+1) & \text{for } i \geq 2 \text{ and } i \leq (I-1) \\ Recv(i+1,i) + Send(i,i+1) & \text{for } i = 1 \\ Recv(i-1,i) + Send(i,i-1) & \text{for } i = I \end{cases}$$

$$\begin{aligned} \text{where } Recv(i,j) &= \textit{time to receive N elements from processor i on processor j} \\ Send(i,j) &= \textit{time to send N elements from processor i to processor j} \\ N &= \textit{number of elements in the dimension not being partitioned} \end{aligned}$$

We can solve the linear system of equations (1) in $O(I^3)$ by simple Gaussian elimination for each A_i. Note, however, that there is no guarantee that each A_i corresponds to an integral number of columns (or rows). To complete the strip decomposition, we must then round the partitions accordingly.

Observe that an alternative, but computationally more complex, solution is to formulate the problem as a constraint-based minimization problem. Linear programming techniques can then be used to derive the partitions. This approach is viable, however in the interest of rapid prototyping, we chose to adopt the simpler linear systems formulation.

2.2 Predicting System State with the Network Weather Service

To solve the linear system of equations (1), we require as parameters the time required to send and receive N elements from each processor to its neighbors (Send(i,j) and Recv(i,j)), and the time required to compute a single element on each processor (P_i).

We can model the send and receive times as

$$\begin{aligned} Send(i,j) &= N * sizeof(element)/Bandwidth(i,j) \\ Recv(i,j) &= N * sizeof(element)/Bandwidth(j,i) \\ \text{where } Bandwidth(i,j) &= \textit{data rate supported by the link between i and j} \end{aligned}$$

Note that N and sizeof(element) are both time-invariant parameters of the problem being solved. Similarly, we can model the per point compute time on each processor i as

$$\begin{aligned}
P_i &= PUnloaded_i/CPU_i \text{ where} \\
PUnloaded_i &= \textit{the time to compute a single point on an unloaded processor } i, \text{ and} \\
CPU_i &= \textit{the percentage of time processor } i \textit{ spends executing partition } i
\end{aligned}$$

These quantities will vary over time due to resource contention. Bandwidth(i,j) will be defined (in part) by the volume and frequency of traffic crossing the link from i to j. CPU_i will depend on the number of additional processes executing on processor i, and the way in which each CPU is managed. Typically, if the system is time shared, the percentage of time a CPU is devoted to any one job is some "fair share" of the total CPU time; however, that share will change as jobs enter and leave the system.

Moreover, the estimates of Send(i,j), Recv(i,j), and P_i must be accurate **at the time the application will be scheduled** which is not necessarily the time at which the partition is derived. The scheduler, therefore, requires a forecast of the values of Send(i,j), Recv(i,j), and P_i for the time frame in which the application will execute.

We have developed a separate facility called the **Network Weather Service** which dynamically supplies values and forecasts for CPU_i for all i, and Bandwidth(i,j) for all i and j in a networked system. The Network Weather Service is outlined in Section 4. For Jacobi2D, the Network Weather Service used dynamic probes and load history to help forecast CPU_i and Bandwidth(i,j) at the time the application was to be scheduled.

2.3 Resource Selection and Scheduling

Resource selection focuses on the identification of a subset of resources that most efficiently support the application. Most users naturally focus on resources they perceive as being "close". For the Jacobi application, we can formally define the logical "distance" between resources and prioritize a resource set based on this metric. Note that **distance between resources is meaningful to the application only in terms of how the resources will be used.** Recall that for a given grid region of size N^2, the computation in each partition scales as $O(N^2)$ and the communication scales as $O(N)$. We can use this relationship to define the **distance** between processors for Jacobi2D. Let

$$\begin{aligned}
P_i &= \textit{the forecast time required for processor } i \textit{ to compute a single point locally} \\
CE(i,j) &= \textit{the forecast time for processor } i \textit{ to send and receive a single element to} \\
&\quad\ \textit{and from processor } j
\end{aligned}$$

Then

$$D(i,j) \quad = \quad N^2 * (|P_i - P_j|) + N * (CE(i,j) + CE(j,i))$$

defines a distance measure between processors i and j for a arbitrary problem size N. Two processors are **near** to each other in Jacobi2D if their compute capabilities are relatively equal, and if their interprocess communication is fast.

To select resources from the global resource pool, we start by identifying a candidate machine to serve as the **locus**. For example, the user's machine or the fastest machine in a cluster may serve as the locus. The rest of the machines are then sorted according to their distance D from the locus. Note that different orderings may be determined for distinct loci. The first K elements of the sorted list for a particular locus L are defined to be the "closest" resource set to L containing K machines.

For Jacobi2D, the workstation with the fastest CPU was used as one such locus. We then used the algorithm in Figure 3 to determine a candidate resource set.

```
let head = locus
let tail = locus
for i in 1 to I-1
        find the machine m such that D(tail,m) is a
            minimum and m is not already on the list
        add m to the tail of the list
        let tail = m
end
```

Figure 3: Prioritizing the resources based on "distance".

```
let locus = machine having the maximum criterion value
let list = a sort of the remaining machines according to
            their logical distance
for k in 0 to I-1
    let S = {locus + the first k elements of list}
    parameterize C_i and P_i for 1 <= i <= |S| with
        Weather Service forecasts
    solve linear system of equations using this parameterization
    if(not all A_i > 0)
        reject partitioning as infeasible
    else if(there exists an A_i that does not fit in free memory
        of processor i)
            reject partitioning as infeasible
        else record expected execution time for subset S
end
implement, the partitioning corresponding to the minimum
        execution time using the S for which it was computed
```

Figure 4: Resource selection and scheduling algorithm for Jacobi2D.

The algorithm iteratively finds the machine that is closest to the current tail, and adds that machine at the tail end of the list. After all I machines have been added, the algorithm terminates with each machine logically closest to those adjacent to it in the list. This form of sorting is useful for a strip decomposition of Jacobi2D as processors only communicate with at most two neighbors.

Having derived the resource list, the Jacobi2D scheduler then proceeds to compare different potential partitionings using subsets of the total list. It starts by estimating the execution time on the locus machine. Next, it considers a two processor partition using the first two processors on the list. It parameterizes the linear system of equations for I=2 processors, and consults the Network Weather Service for the performance forecasts that pertain to those two machines. After solving the linear system, it records the estimated execution time of the resulting partition. A three processor partitioning using the first three processors from the sorted resource list is considered next. The estimated execution time for the three processor system is recorded, and the algorithm continues until all processors from the list are considered or some predefined maximum logical distance from the locus is reached. Finally, a processor set and a partitioning and schedule yielding the minimum estimated execution time are chosen as the "best" schedule for that locus. Note that when I=1, the Resource Selector considers a single-site implementation. In our example, the single-site implementation is simply a sequential version of the KeLP implementation. If an optimized implementation for a particular system were available, the Resource Selector could consider that as well.

Each time a partition is generated in the process, it is checked for feasibility. Two **filters** are employed to remove infeasible partitions from those ultimately considered for scheduling. The first filter removes partitions that have negative values of A_i. These correspond to mappings where the communication time is so great, the processor must compute a negative number of elements (implying a negative execution time) in order to finish with the other processors. The second filter checks to make sure that the size of each partition fits within the free memory (forecast by the Network Weather Service) available on the machine to which it is assigned.

The resource selection and scheduling method used by our example Jacobi2D scheduler can be summarized by the pseudocode in Figure 4.

2.4 Scheduling Jacobi2D and the AppLeS PrincipLeS

The scheduling approach we have described for Jacobi2D uses the principles outlined in Section 1.1 and in fact is an example of an AppLeS. Application-specific and system-specific information are used throughout the scheduler, both to generate schedules and to select resources. Dynamic system information is provided via the Network Weather Service to parameterize performance models. Predictive models are used to evaluate and rank candidate schedules. Finally and perhaps most important, all resources are considered strictly in terms of how they affect application performance.

Using this application-level approach to scheduling, the natural question becomes "How performance-efficient is the schedule that it generates?" We describe experiments which address this question in the next section.

3 Performance Results for the Jacobi2D Application-Level Schedule

To determine the effectiveness of the application-level scheduling approach, it is important to answer the following questions:

- How does the execution time of Jacobi2D using an AppLeS schedule compare to a schedule determined using a widely-accepted conventional method?
- What is the effect of using dynamically forecast resource performance data in the application-level scheduling approach?
- What is the effect of automatic resource selection in the application-level scheduling approach?

To address these questions, we compared four partitioning methods for the same KeLP implementation of Jacobi2D. The first method [**Compile-time blocked**] uses a conventional HPF-style [14] block partitioning in which each processor is assigned (at compile-time) a relatively equal-sized square region of the grid to compute. The other three partitioning methods utilize versions of the application-level scheduling approach described in the previous section. Partitioning method 2 [**Compile-time AppLeS**] uses good static estimates for resource performance and uses resource selection to select a resource set from the total resources. Partitioning method 3 [**Runtime AppLeS/No Select**] uses dynamic estimates from the Network Weather Service for resource performance but assumes that the user wants to use **all** available resources. Partitioning method 4 [**Runtime AppLeS**] uses dynamic estimates and resource selection – it constitutes the full application-level scheduling approach discussed in the last section. Note that partitioning methods 3 and 4 utilize Network Weather Service data and so must be performed at run-time, whereas partitioning methods 1 and 2 use static data and may be performed at compile-time.

All four versions first partition and distribute the grid, and then execute the Jacobi solver. That is, the data and computations are scheduled on the processors once before execution begins, and remain there for the duration of the execution. We are currently formulating a version of the Jacobi application-level scheduler which effectively redistributes the grid in response to changing load on system resources. This flexibility is supported in the AppLeS software described in the next section.

3.1 Execution Performance

To investigate the relative execution performance of the four partitioning methods, we used eight non-dedicated workstations located at the San Diego Supercomputer Center (SDSC) and the U.C. San Diego Parallel Computation Laboratory (UCSD-PCL). The workstation set consisted of a Sun Sparc-2, a Sun Sparc-10, and two IBM RS6000 workstations located at UCSD, and four DEC Alpha workstations located at SDSC. Numeric format conversions were handled by KeLP which uses MPI as its underlying communication substrate. The network connecting these systems was also heterogeneous and non-dedicated. Within the

PCL, the Suns were attached to an ethernet segment shared by several other systems. The RS6000s were connected to a different segment (also shared by other ambient machines) and a gateway which linked the two segments. At SDSC, the Alpha workstations were connected to non-dedicated FDDI ring. The configuration is shown in Figure 5.

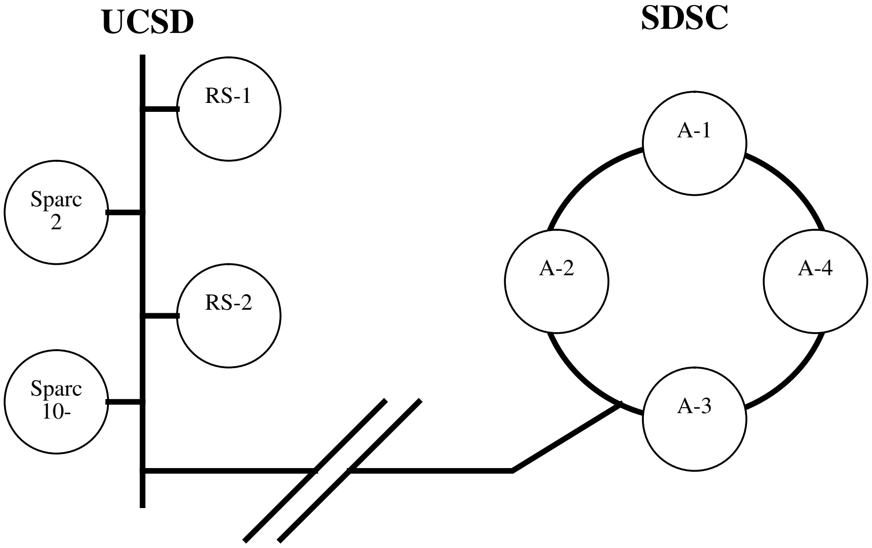

Figure 5: Workstations and Networks used at UCSD and SDSC.

All systems and networks were shared and used in "production mode" while we ran our experiments. Since conditions might change between one execution and the next (due to contention) we made several runs for each problem size, and reported the average execution time of a single iteration. During each experiment, we ran one instance of each of the four partitioning methods back-to-back hoping that all four executions would enjoy similar conditions, on average. Figure 6 shows the average iteration execution times (in seconds) for a range of problem sizes. In each case, a square grid having the problem size dimension shown in the figure was used.

In the experiments, application-level scheduling is able to outperform the block partitioning because it uses its performance model to **predict** how well each resource will perform when executing a piece of Jacobi2D. It uses that prediction to determine how much of the grid should be assigned to each machine. Notice also, that the benefit gained from using dynamic performance forecasts is substantial. Less obvious, however, is the improvement gained through resource selection. While the version that used resource selection does run between 25% and 50% faster than the non-selecting runtime AppLeS, the relative improvement compared to the blocked implementation is not large. However, the range of feasible partitions for the non-selecting runtime AppLeS is limited. For example, under the conditions during which the experiments were conducted, it was not possible to balance the execution time for a 500 by 500 element problem: the communication delay between UCSD and SDSC was so great that processors in either end would need to compute for a negative amount of time to compensate.

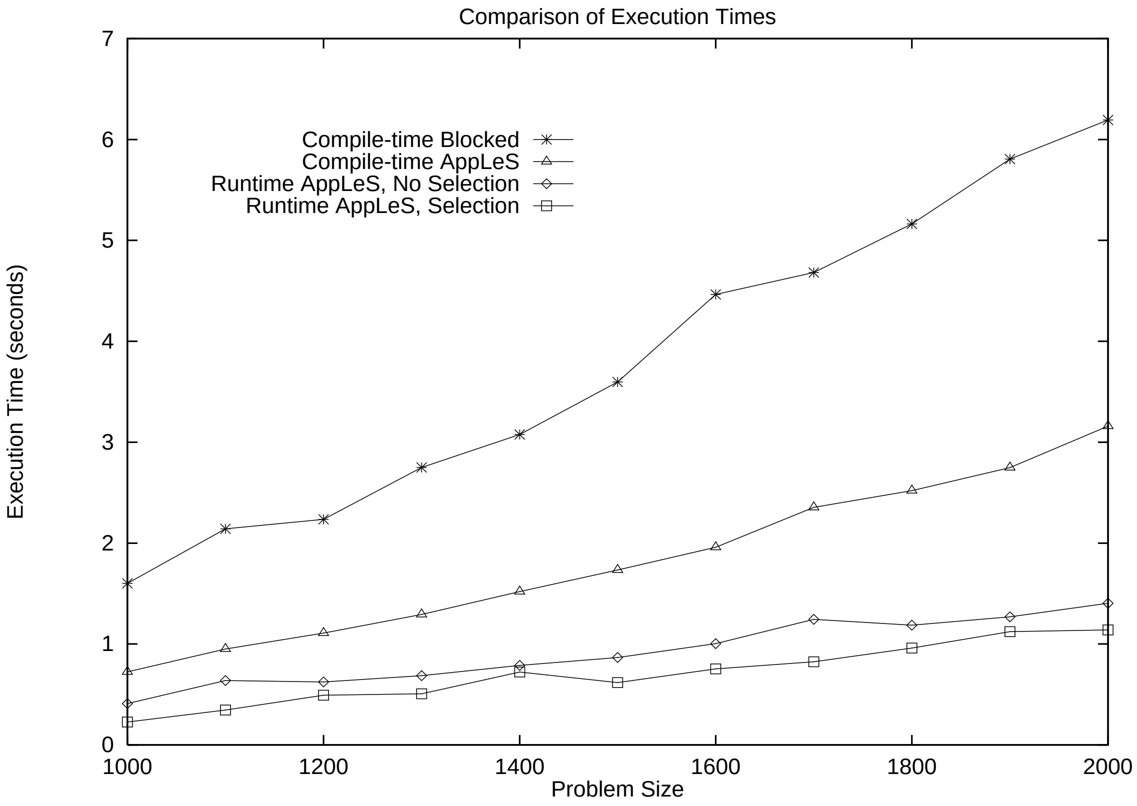

Figure 6: Execution times for Jacobi2D.

In Figure 7 we show execution time data for a wider range of problem sizes using Compile-time Blocked, and the full AppLeS partitioner. Without resource selection, AppLeS would only be able to compute reliably (depending on contention conditions) over the 1000 to 2000 problem size domain. We also show the predicted execution time AppLeS computed for each run. For each problem size, we plot the time that the performance model predicted against the actual execution time that resulted for each mapping. It is the accuracy of the performance model that allows AppLeS to choose good resource mappings.

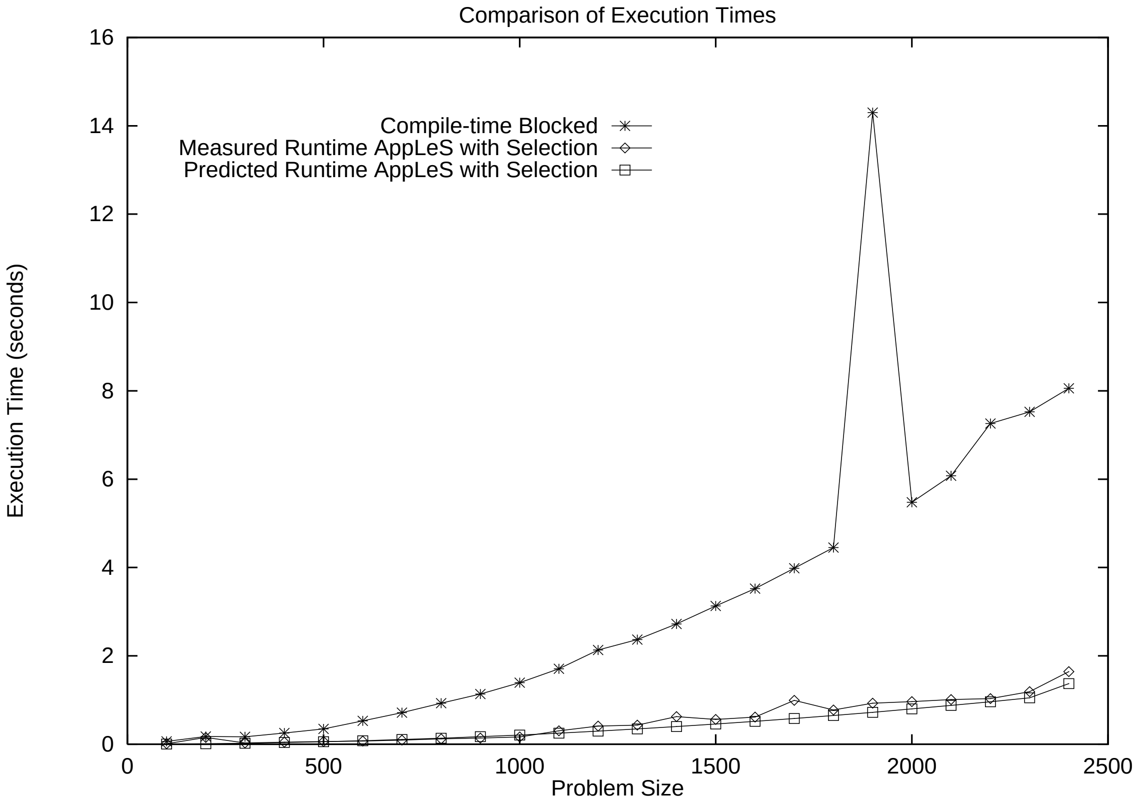

Figure 7: Execution times for Jacobi2D.

Note also the large spike in execution time for the blocked partitioning at the 1900 problem size abscissa. During one experimental run at that size, a network gateway between UCSD and SDSC went down forcing all communications between the two to use an alternative and much slower route. The AppLeS agent (through Network Weather Service readings) was able to detect the sudden drop in available bandwidth and avoid partitionings that spanned the affected link.

3.2 Partitioning for Memory Availability

Distributed parallel execution also allows an application to aggregate memory resources so that problems that are larger than will fit into any single memory may be solved. Indeed, the motivation behind the parallel implementation of many codes stems from the need to use collections of memory systems rather than a desire for concurrent execution. To investigate the ability of the AppLeS approach to effectively aggregate memory, we added to the resource pool two IBM SP-2 processors with 128 megabytes of real memory each. The SP-2 uses virtual memory on each of its nodes so that more than 128 megabytes of memory may be used. However, memory is paged to disk causing reference times to increase dramatically when the real memory of the system is exceeded. During the experiments, we had dedicated access to the two SP-2 processors and the link between them, but they were connected to the rest of the resources via a shared ethernet segment. Figure 8 shows the resource pool including the SP-2 nodes.

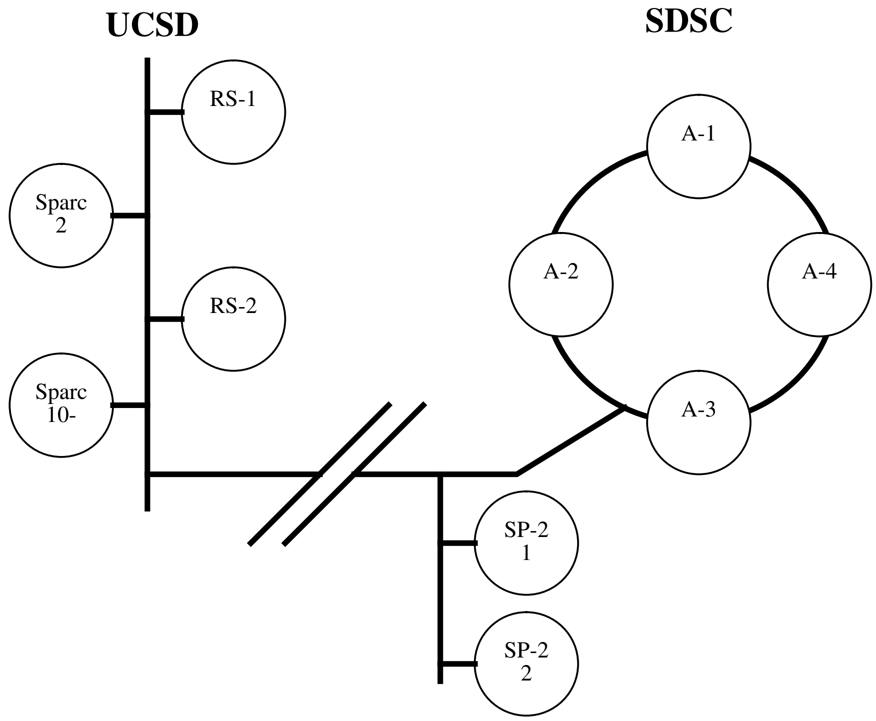

Figure 8: Resource Pool Including SP-2 Processors.

Since the processors were completely unloaded, and their connectivity to the other resources suffered from contention, the best partitioning (yielding the shortest execution time) was to split the grid evenly between the two SP-2 nodes as long as neither partition exceeded the available real memory on each node. However, when the problem size caused the partitions to spill out of the available real memory, the resulting delays due to paging caused execution time to increase substantially. In Figure 9 we show the execution time of a blocked partitioning using the SP-2 processors only versus the AppLeS approach for Jacobi2D.

For problem sizes less than 3900 by 3900, AppLeS correctly chose the mapping using the SP-2 processors and exhibited nearly identical execution times to the blocked mapping. As problem size increased, the SP-2 began paging, causing execution time to increase to the

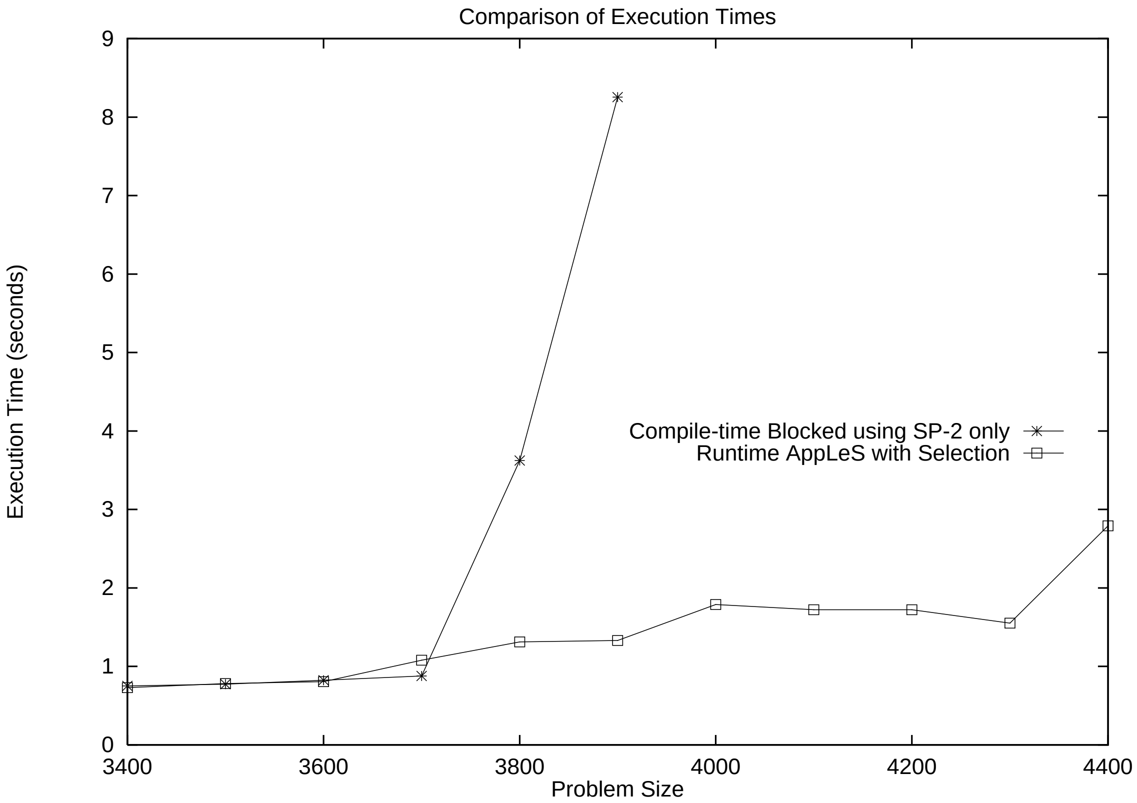

Figure 9: Partitioning and Memory Usage.

point where use of these processors was no longer feasible. The AppLeS agent was able to locate memory elsewhere within the resource pool effectively. At each problem size beyond 3900, the AppLeS was able to find memory it could use effectively without a dramatic change in the performance trajectory.

Thus far we have shown how the AppLeS approach was used effectively to determine a performance-efficient (and non-obvious) schedule for Jacobi2D. It was important to walk through this example in detail to demonstrate this approach. We now discuss how the AppLeS approach can be used as the basis for the design of general software agents which facilitate application-level scheduling for distributed parallel applications.

4 Developing General AppLeS Agents

It is clear from the previous sections that application-level scheduling can be used effectively to achieve performance for distributed applications. However, to develop general AppLeS agents, we must convince ourselves that the following questions could be answered in the affirmative:

- Is the application-level approach for selecting a performance-efficient schedule generalizable?

- Is it possible to efficiently obtain the appropriate level of application and system information (from the user or through analysis) from which good schedules may be derived?

To address the first question, observe that in the development of the application-level schedule for Jacobi2D, our approach did not rely particularly on the choice of algorithm, implementation language, or programming style for success. The organization of the AppLeS software mimics how a diligent user would schedule his or her application. The characteristics of the application are relevant only as they pertain to modeling its performance. In AppLeS, we modularize application-specific, system-specific and dynamic information and use this information to parameterize the general approach.

To address the second question, we developed a set of data sources to provide the relevant application- and system-specific information efficiently. The Network Weather Service was designed to provide dynamic system information and short-term forecasts. Application-specific information is provided through a Heterogeneous Application Template (or HAT) which distills much of the information from the application relevant to performance estimation. Additional information which reflects the user's preferences, access to resources, etc. is provided by User Specifications. Note that for AppLeS, as in practice, the more complete the application information that is available to the scheduler, the better the schedule.

AppLeS is currently a work-in-progress. The software has been designed and the underlying building blocks are currently being prototyped. We are working with researchers from the Legion project [12], [17] and from the Globus project [3], [11] to prototype AppLeS as an application-level scheduler for these resource management systems. In addition, we are progressing on an implementation which uses MPI as the underlying substrate.

Note that AppLeS essentially develops a customized scheduler for each application. This differs from the approach taken in much of the scheduling literature ([21], [13], [19], [23] [7] etc.). Application-level scheduling is related to the work of Brewer [2], and more directly to the Mars project [8]. Brewer's work, which attempts to select the correct implementation of an algorithm for a given machine based on a small set of static parameters, uses application-specific information to improve performance. The MARS project [8], whose goal is to produce more general-purpose software, is more similar in scope and intent to AppLeS. An important difference, however, is that AppLeS includes **user-specific** as well as application-specific information in its scheduling decisions. User-specific information provides a powerful and well-defined interface that allows the user to influence and control how the scheduling agent will behave.

In the following sections, we describe the architecture for general AppLeS agents.

4.1 The AppLeS Organization

AppLeS is organized in terms of four subsystems and a single active agent called the **Coordinator**. The four subsystems are

- **The Resource Selector** which chooses and filters different resource combinations for the application's execution,
- **The Planner** which generates a description of a system-independent schedule from a resource combination,
- **The Performance Estimator** which generates a performance estimate for candidate schedules according to the user's performance metric, and
- **The Actuator** which implements the "best" schedule on the target resource management system(s).

Figure 10 depicts the Coordinator and these four subsystems. Application-specific, system-specific, and dynamic information used by these subsystems constitute an "information pool" which all subsystems share. There are four general sources of information feeding the information pool. The **Network Weather Service** provides dynamic information on system state and forecasts of system state for the time frame in which the application will be scheduled. The **Heterogeneous Application Template** is a web-oriented interface in which the user provides specific information about the structure, characteristics and current implementations of the application and its tasks. The **User Specifications** provide information on the user's criteria for performance, preferences for implementation, additional application information, etc. Finally, the **Model** pool provides model templates used by the AppLeS subsystems for application performance estimation.

AppLeS agents will be employed as follows: Initially, the user provides information to the agent via the HAT and User Specifications. The agent uses the Resource Selector to select a set of viable resource configurations based on accessibility, the user's access rights, the characteristics of the application (input as filters which exclude resources that are not viable), and a notion of "distance" which is derived from HAT information and the Model pool, or provided as a default by the Coordinator. For each viable resource configuration, the Planner (in conjunction with the Performance Estimator and the Network Weather Service)

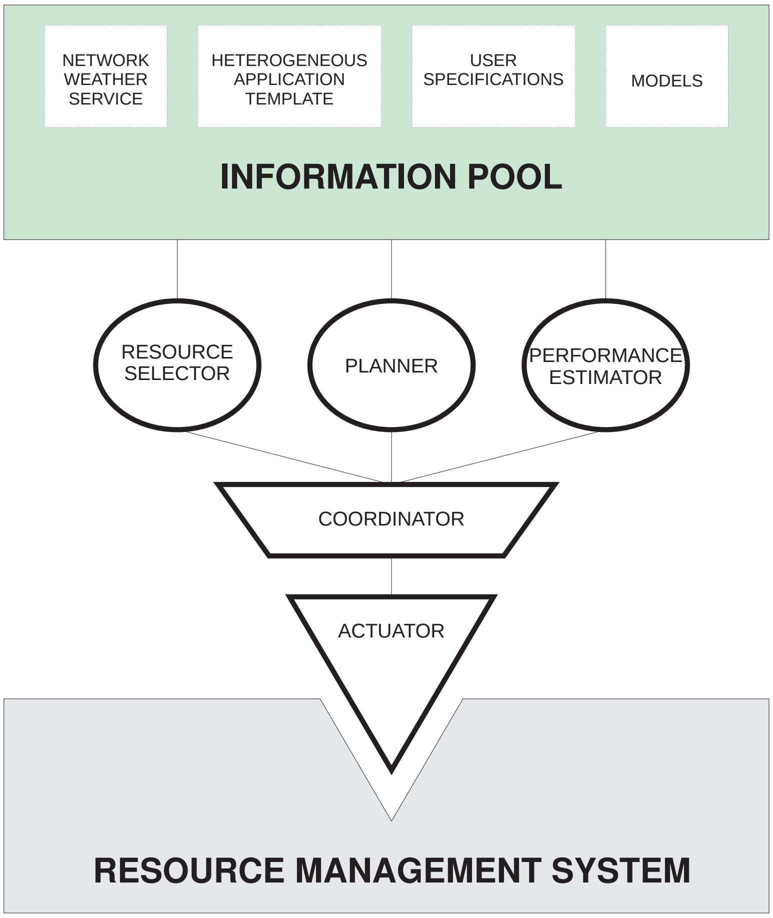

Figure 10: Relationship of the components of AppLeS.

computes a potential schedule of the resources using predictive models from the Model pool. The Coordinator considers the performance of the candidate schedules and selects a "best" schedule for implementation. The Actuator then interacts with the resource management system(s) to implement this schedule.

In the following subsections, we describe each of the components of AppLeS agents in more detail.

4.2 The Coordinator

The Coordinator embodies the active thread or threads of control within an AppLeS agent. It executes a **blueprint** that dictates the way in which it uses the various other subsystems to derive a schedule, initiate the application, and monitor its progress. The blueprint can be specified by the user or by the system for a particular application or class of applications (e.g. data parallel applications). We show a sample blueprint in Figure 11. This is typical for a user scheduling a minimum execution time application over a large set of possible resources,

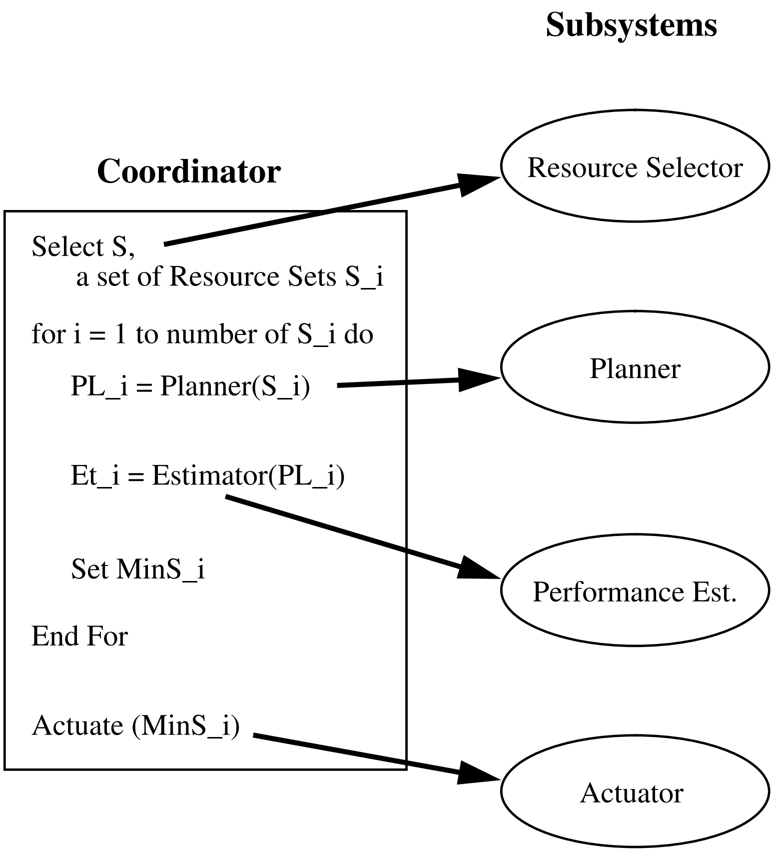

Figure 11: Coordinator and Blueprint

and is, in fact, the blueprint used to schedule Jacobi2D.

4.3 Data Sources

While the Coordinator directs the interactions between subsystems through its blueprint, each subsystem draws upon a variety of data sources to perform its function. These data sources contribute information to a data pool which is available to all AppLeS functions (see Figure 10). They are the **Network Weather Service** (NWS), the **Heterogeneous Application Template** (HAT), the **User Specifications**, and the **Model** pool. In this section, we briefly describe the form and content of each.

4.3.1 The Network Weather Service

The Network Weather Service provides software for monitoring and predicting the load (or "weather") on networked resources. Our strategy is to use sensors to dynamically probe and read the network "weather" conditions such as CPU load, available free memory, network performance, etc.

To provide forecasts of system state, the Network Weather Service uses a number of stochastic techniques for predicting network load. Experiments using different network links and predictors show that, in general, for a given resource, different estimation techniques will yield the best forecasts at different times. Consequently, the Network Weather Service tracks the error between all predictors and sampled data, and uses the predictor with the lowest cumulative error to make predictions of system state. Both the prediction and a measure of its recent "accuracy" are used by the Resource Selector, the Planner, and the Performance Estimator subsystems of an AppLeS agent.

We have prototyped this facility with good results as shown in the Jacobi2D example. We are currently integrating Network Weather Service facilities with the Legion and Globus resource management systems.

4.3.2 The Heterogeneous Application Template

The Heterogeneous Application Template (HAT) provides basic information about the overall application, tasks and implementations in terms of their resource requirements. Information is provided through a web interface which makes explicit the structural parameters of the application, information about existing implementations of application tasks, and the data movement requirements between distinct tasks. Figures 12, 13 and 14 give a sample of HAT parameters.

The HAT also lets the user identify an **active set**, i.e. a set of task/machine implementations that work together to compose an entire application. Since there may be multiple implementations, the active set identifies the particular task/machine allocations that will be used in a single full implementation of the application. For Jacobi2D, the active set was composed of a single task implementation per machine. In general, however, there may be several implementations from which to choose and multiple active sets.

Notice that the user may not have all the information requested by HAT. The system can use partial information to determine a schedule. However, as is the case for the user, the better and more comprehensive the information available, the more performance-efficient the schedule is likely to be.

4.3.3 User Specifications

While the HAT describes application-specific information, information specific to a particular user or application developer is made available to an AppLeS through User Specifications (US) which will also be html-based. The most important role of the US is in the definition of user-specified requirements which fall into the three broad categories: execution constraints, performance objectives, and user preferences. **Execution constraints** refer to the access rights and resource constraints of the user. The user's **performance objective** is also conveyed through the US. For Jacobi2D, minimum execution time was the desired objective. Finally, the US allows the user to specify **preferences** for the Coordinator to attempt to satisfy. It may be that one resource should be preferred over another for non-performance related reasons. This feature gives the user tremendous control over the actions of AppLeS and the solutions that it generates.

4.3.4 Models

The Model pool provides a set of model templates which are used for application performance estimation by the Planner, Performance Estimator and Resource Selector. Model templates are structures for composing models of characteristics which contribute to application performance. For example, in Jacobi, the model template for the execution time for processor i is

$$T_i \;=\; \mathit{Computation} + \mathit{Communication}$$

where Computation is instantiated as $A_i * P_i$ and Communication is instantiated as C_i, as described in Section 2.1.

Model templates may be provided by the user. Default model templates for classes of applications (e.g. data parallel regular grid applications) will be available in the Model pool. Note that model templates can leverage successful models from the literature such as [2], [18] [10], [24], [22], etc. to predict the performance of the application and its tasks.

4.4 Resource Selector

The Resource Selector produces viable active sets to be considered by the Coordinator. It may iterate multiple times to identify a set of candidate active sets according to different selection criteria.

A potentially viable active set may be **filtered** to ensure its feasibility. Resources are prioritized with respect to an application-specific valuation function such as "distance", and filters are applied to the resource set to eliminate resources that are unusable. A filter may use information such as the user's access rights, memory constraints, implementation availability, etc. to eliminate resources quickly. Viable and feasible resource configurations will be scheduled by the Planner, evaluated by the Performance Estimator, and compared by the Coordinator to other candidate schedules.

In the Jacobi2D example, filters considered two characteristics of each potential schedule: the area of region i, A_i, and the available memory. Partitions with strips in which A_i was negative were filtered out. Next, resources which did not meet the memory requirements of application tasks were also filtered out. Such constraints for most users are readily identifiable, and can be used profitably to reduce the resource selection space.

4.5 Planner

The function of the Planner is to create a schedule for a feasible active set. The schedule is based on a scheduling policy that optimizes for the user's performance measure. In practice, most users will employ common performance measures (execution time, cost, speedup), and the Planner will be equipped with default scheduling policies for these measures if the user chooses not to recommend a policy of his/her own. The schedule generated by the Planner must be in a format that the Actuator (described in section 4.7) can implement on the target resource system.

In the Jacobi2D example, the Planner implemented a time-balancing scheduling policy. It took a list of candidate machines and their communication links (the feasible resource set), and produced a mapping of grid strips to the machines. The Coordinator then used the Performance Estimator to determine the execution time of each mapping generated by the Planner and passed the best schedule to the Actuator.

4.6 The Performance Estimator

The performance estimator parameterizes a model template with component models to produce an estimate of application performance, given a schedule provided by the Planner. Parameters for the component models can be provided by the user or derived from other data sources such as the Network Weather Service. Since dynamic information is included, the resulting estimates can be targeted to the time frame during which the application will be run by the Actuator. In Jacobi, the formula $T_i = A_i * P_i + C_i$ is evaluated to obtain an estimate of the time necessary to compute each strip.

Note that it is important to estimate the behavior of the application tasks in the context of the production systems in which they will be used. For this reason, we are developing models which forecast the **slowdown** of tasks on shared resources (networks and machines) [4]. Factoring slowdown into the model will provide a more realistic estimate of application and task performance in the presence of contention.

4.7 Actuator

AppLeS does not function as a resource manager – it relies on the services of existing resource managers to perform resource allocation and task instantiation. It is the job of the Actuator to implement the schedule (determined by the Planner) using the semantics and facilities supported by the target resource management system. Some of these resource managers, such as PVM, are limited in scope and provide little additional functionality. Others, like Legion, have the potential for communicating considerable information about resource and application status. The Actuator will also convey whatever feedback information is available to the various subsystems. It acts as the conduit between the Coordinator and the underlying resource management facilities.

The minimum functionality required by the Actuator is the ability to initiate a network connected task on a remote machine. More accurate scheduling can be accomplished when the resource management system returns feedback about when resources are actually available for use, or can provide guaranteed service times in response to requests for service. Since the AppLeS agent is working at the application level, however, the Actuator minimally has access to whatever facilities the application enjoys. It will use the same facilities to communicate with the application and manage its task execution that the application itself uses to control its tasks. In that sense, the Actuator, and by extension the AppLeS agent, constitute an integrated extension of the program being scheduled. AppLeS and the application become part of the same execution instance. In the Jacobi2D example, the Actuator issued KeLP directives to control grid partitioning. These were the same primitives the application used to manage the grid itself.

5 Summary

As network speeds increase and parallel distributed computing becomes more prevalent, resource-intensive applications will increasingly need to leverage shared, heterogeneous networks of resources. Effective coordination of application components and their use of resources is key to performance. In this work, we described **application-level scheduling** as a way of achieving performance-efficient schedules for applications which execute on heterogeneous networks of machines. We described principles which reflect the way in which applications are scheduled by their end-users and illustrated these principles by developing a "proof-of-concept" application-level scheduler for a distributed data-parallel Jacobi application. We then described a general architecture for Application-Level Schedulers and described the subsystems which compose an AppLeS agent.

From the results generated by our prototype, it is clear that the AppLeS approach can achieve substantial performance improvements for an individual application over conventional scheduling methods. Application-level scheduling allows the user to deal with the heterogeneous system as it really is: under the control of multiple system schedulers, shared by other contending applications, and able to deliver only a dynamically varying fraction of resource performance. When such characteristics are explicitly factored into the scheduling activity, the application can better leverage the system to achieve performance.

Acknowledgments

We are grateful to the researchers in the UCSD Parallel Computation Laboratory, and in particular to Stephen J. Fink for many substantive discussions. We are also grateful to Andrew Grimshaw, Carl Kesselman, Reagan Moore, John Karpovich, Doug Shea, and Darren Atkinson for their thoughtful comments and support.

AppLeS Home Page

`http://www-cse.ucsd.edu/users/berman/apples.html`

References

[1] Berman, F. and Moore, R., **Heterogeneous working group report**, Proceedings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments, 1995. `http://cesdis.gsfc.nasa.gov/PAS2.index.html`.

[2] Brewer, E. A., **High-level optimization via automated statistical modeling**, Proceedings of Principles and Practice of Parallel Programming, PPoPP'95 (1995), pp. 80–91.

[3] DeFanti, T., Foster, I., Papka, M., Stevens, R. and Kuhfuss, T., **Overview of the I-way: Wide area visual supercomputing**, to appear in the International Journal of Supercomputer Applications.

[4] Figueira, S. M. and Berman, F., **Modeling the effects of contention on the performance of heterogeneous applications**, to appear in the Proceedings of the High Performance Distributed Computing Conference (1996).

[5] Fink, S., `http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html`, 1995.

[6] Fink, S. J., Baden, S. B. and Kohn, S. R., **Flexible communication mechanisms for dynamic structured applications**, in preparation, 1996.

[7] Freund, R., Proceedings of the 1996 IPPS Workshop on Heterogeneous Computing.

[8] Gehrinf, J. and Reinfeld, A., **Mars - a framework for minimizing the job execution time in a metacomputing environment**, Proceedings of Future general Computer Systems (1996).

[9] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., **PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing**, MIT Press, 1994.

[10] Getov, V. S., Hockney, R. W. and Hey, A. J. G., **Performance analysis of distributed applications by suitability functions**, in Proceedings of the MPPM Conference (1993).

[11] **Globus**, `http://www.mcs.anl.gov/globus`.

[12] Grimshaw, A. S., Wulf, W. A., French, J. C., Weaver, A. C. and Reynolds, P. F., **Legion: The next logical step toward a nationwide virtual computer**, Tech. Rep. CS-94-21, University of Virginia, 1994.

[13] Hensgen, D. A., Moore, L., Kidd, T., Freund, R., Keith, E., Kussow, M., Lima, J. and Campbell, M., **Adding rescheduling to and integrating condor with smartnet**, in Proceedings of the Heterogeneous workshop (1995).

[14] High Performance Fortran Forum, **High performance fortran language specification**, Rice Univeristy, Houston, Texas, May 1993.

[15] Hoffman, J. D., **Numerical Methods for Engineers and Scientists**, McGraw-Hill , Inc, 1992.

[16] Korab, H. and Brown, M., **Virtual environments and distributed computing at SC'95: GII testbed and HPC challenge applications on the I-way**. in Proceedings of Supercomputing '95 (1995).

[17] **Legion**, `http://www.cs.virginia.edu/~mentat/legion/legion.html`.

[18] Messina, P. and Heirich, A., personal communication, 1995.

[19] Pruyne, J. and Livny, M., **Parallel processing on dynamic resources with Carmi**, in Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, IPPS '95 (April 1995).

[20] **PVM**, `http://www.epm.ornl.gov:80/pvm/`.

[21] Rudolph, L. and Feitelson, D., Proceedings of the 1996 IPPS Workshop on Job Scheduling Strategies for Parallel Processing (1996).

[22] Sarkar, V., **Automatic partitioning of a program dependence graph into parallel tasks**, IBM Journal of Research and Development 35, 5/6 (Sept/Nov 1991).

[23] Siegel, H., Antonio, J., Metzger, R., Tan, M. and Li, Y. A., **Heterogeneous computing**. Tech. Rep., Purdue University EE Technical Report TR-EE 94-37.

[24] Zhang, X. and Yan, Y., **A framework of performance prediction of parallel computing nondedicated heterogeneous now**, in Proceedings of the 1995 International Conference on Parallel Processing (1995), pp. 163–7.

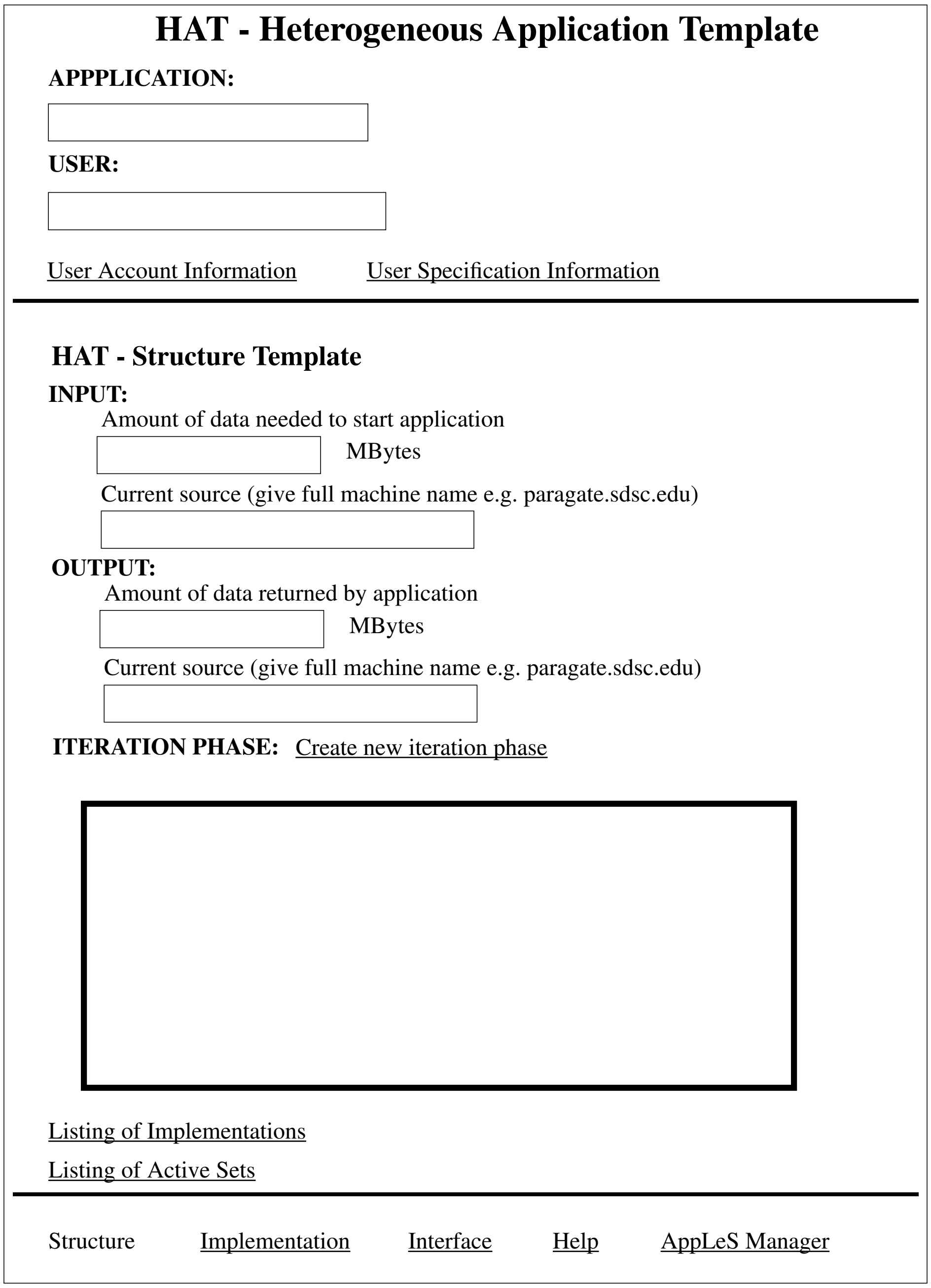

Figure 12: The Structure module of HAT gives information about the general functional decomposition of the application, and lets a user identify an **active set** for the application.

HAT - Implementation Template

TASK:

PLATFORM:

PARADIGM:

Sequential Vector Task Parallel Data Parallel

Single Processor Multi-Processor

USAGE:

Dedicated Non-dedicated

DATA STRUCTURES:

Number

Size Bytes

Compution per data structure MFlops

Communication per data structure Words

RATIO:

Select an approximation or fill in numerical values

Communication Heavy Balanced Computation Heavy

Compution per data structure MFlops

Communication per data structure Words

COMMUNICATION PATTERNS:

Pt to Pt Stencil Multicast Broadcast

MEMORY:

Core memory needed for in-core sol'n MWords

TUNING FACTOR:

1 (bubblesort) 3 (cs 101) 5 (1st year grad) 7(PhD thesis) 10(hand tuned asseumbler)

Structure Implementation Interface Help AppLeS Manager

Figure 13: The Implementation module focuses on how the task was implemented for a specific platform.

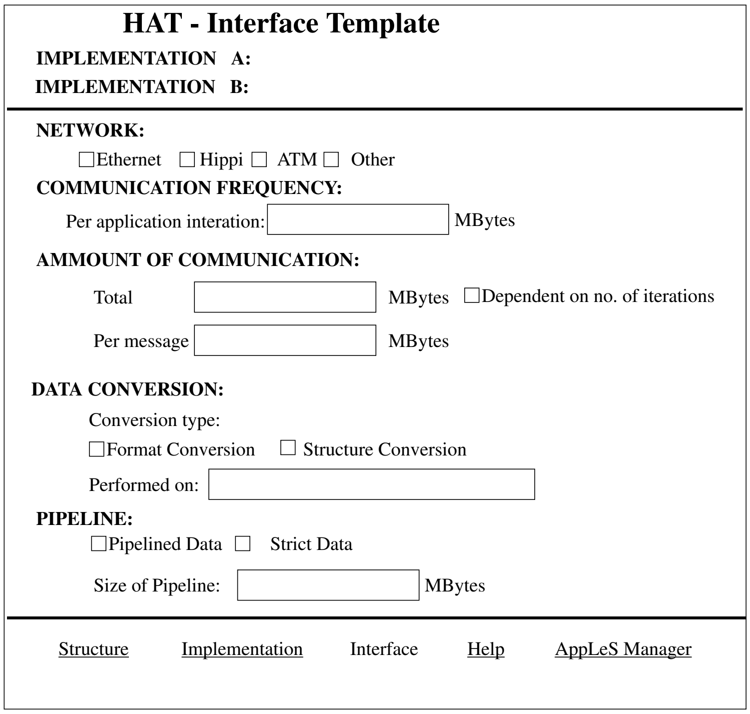

Figure 14: The Interface module of HAT characterizes the communication between implementations A and B mapped to distinct execution sites.