Exploiting Application Tunability for Efficient, Predictable Parallel Resource Management

Fangzhe Chang, Vijay Karamcheti, and Zvi Kedem
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*fangzhe,vijayk,kedem*}*@cs.nyu.edu*

Abstract

Parallel computing is becoming increasing central and mainstream, driven both by the widespread availability of commodity SMP and high-performance cluster platforms, as well as the growing use of parallelism in general-purpose applications such as image recognition, virtual reality, and media processing. In addition to performance requirements, the latter computations impose soft real-time constraints, necessitating *efficient, predictable parallel resource management*. Unfortunately, traditional resource management approaches in both parallel and real-time systems are inadequate for meeting this objective; the parallel approaches focus primarily on improving application performance and/or system utilization at the cost of arbitrarily delaying a given application, while the real-time approaches are overly conservative sacrificing system utilization in order to meet application deadlines.

In this paper, we propose a novel approach for increasing parallel system utilization while meeting application soft real-time deadlines. Our approach exploits the *application tunability* found in several general-purpose computations. Tunability refers to an application's ability to trade off resource requirements over time, while maintaining a desired level of output quality. In other words, a large allocation of resources in one stage of the computation's lifetime may compensate, in a parameterizable manner, for a smaller allocation in another stage. We first describe language extensions to support tunability in the Calypso programming system, a component of the MILAN metacomputing project, and evaluate their expressiveness using an image processing application. We then characterize the performance benefits of tunability, using a synthetic task system to systematically identify its benefits and shortcomings. Our results are very encouraging: application tunability is convenient to express, and can significantly improve parallel system utilization for computations with predictability requirements.

1 Introduction

Parallel computing has become central and mainstream, driven both by a decrease in cost and a growth in its general-purpose applicability. The commodity nature of small-scale symmetric multiprocessors (SMPs) and high-performance cluster platforms [4] has ensured the widespread availability of relatively inexpensive hardware platforms. Moreover, these platforms are increasingly being used to run general-purpose applications such as image recognition, virtual reality, and media processing [6]. However, these latter applications introduce additional demands on the management of resources in a parallel system. On top of the performance requirements traditionally associated with parallel applications (e.g., floating-point intensive scientific codes), these applications impose additional soft real-time constraints requiring completion of different portions of the application within specific intervals of time. For example, an application that is trying to analyze a live video feed to recognize important artifacts for archival and classification purposes needs to complete its processing by the time the next frame arrives. Thus, such applications necessitate *efficient, predictable parallel resource management* that must simultaneously optimize resource utilization and ensure that applications meet their deadlines.[1]

Unfortunately, traditional resource management approaches in both parallel and real-time systems are inadequate for meeting these objectives. The parallel approaches focus primarily on improving application performance and/or system utilization at the cost of providing only *best effort* guarantees to the application. A specific application can experience arbitrary delay which may grow with the number of applications contending for the resources. Clearly, such delays are unacceptable for soft real-time applications that work with continuous media. Real-time systems are better at providing predictable guarantees to applications. However, they do so by being overly conservative, ensuring that enough resources are available for each application to meet its deadline. Admission control is used to ensure that the system is always underloaded, providing application predictability at the cost of system utilization.

In this paper, we propose a novel approach for increasing parallel system utilization while meeting application soft real-time deadlines. Our approach exploits the *application tunability* found in several general-purpose computations such as the ones described above. Tunability refers to an application's ability to trade off resource requirements over time, while maintaining a desired level of output quality. In other words, a large allocation of resources in one stage of the computation's lifetime may compensate, in a parameterizable manner, for a smaller allocation in another stage. For example, the artifact recognition application may first sample different portions of the image to decide on interesting regions, and then run a resource intensive algorithm on these regions; spending more resources on the sampling step reduces the work that will need to be performed in the analysis step. Application tunability provides flexibility to the underlying resource management system, which can now use the choice in resource allocation profiles to increase the number of applications that can be admitted into the system (improving utilization), while still ensuring that an application meets its real-time requirements.

This paper describes support for predictable, tunable applications in the MILAN parallel and distributed computing system [17]. We first describe language extensions to the Calypso programming language for expressing application tunability, and evaluate their effectiveness by describing the construction of a tunable image processing application (junction detection). We then characterize the performance benefits of application tunability, by first proposing a competitive heuristic for the underlying scheduling problem, and then using a synthetic task system to systematically identify the benefits and shortcomings of tunability. We find that tunability helps improve both system utilization and job throughput (the number of jobs which meet their deadlines) over a wide range of task parameters, such as arrival rate, deadline laxity, and the shape of the required resource profile. Our results are very encouraging: application tunability is convenient to express, and can significantly improve parallel system utilization for computations with predictability requirements.

The rest of this paper is organized as follows. Section 2 provides relevant background on the MILAN system and the Calypso programming language. In Section 3, we present the overall MILAN resource management architecture for predictable computations. Section 4 describes extensions to the Calypso source language for supporting tunability. The performance impact of tunability is characterized in Section 5. Section 6 places our work in context with related efforts and we conclude in Section 7.

[1]In this paper we assume that processors are the primary resource that is being managed: applications request non-preemptive allocation of a specific number of processors for a fixed amount of time.

2 Background and Project Context

We describe, in turn, the MILAN parallel and distributed computing system, and the Calypso programming language. Application tunability is expressed employing extensions to the Calypso language, and is exploited by the MILAN resource manager.

2.1 The MILAN System

The MILAN system supports robust, predictable parallel computation over a possibly heterogenous collection of resources. MILAN takes advantage of two execution techniques with strong theoretical foundations [5]—*two-phase idempotent execution strategy*, and *eager scheduling*—to provide programmers with the view of a fault-free virtual shared memory environment, even when the underlying resources may incur faults and exhibit wide variations in processing speeds. This support is exposed to the programmer in the form of several programming systems: Calypso [1] described in further detail below, Chime [15] which supports distributed execution of CC++ [3] programs, and Charlotte [2] which provides a web-based metacomputing infrastructure. In addition, the MILAN system consists of supporting infrastructure such as ResourceBroker (a system for dynamically managing the association and integration of resources into multiple parallel computations according to user-specified policies) and Knitting Factory (a toolkit for construction of distributed applications in an unpredictable metacomputing environment).

The MILAN resource manager allocates system resources among competing applications based upon their resource and predictability requirements, and provides the context for this research. Section 3 describes the resource manager in greater detail.

2.2 The Calypso Programming Language

The programming model supported by Calypso is one of parallel tasks inserted into a sequential program. These parallel tasks are responsible for performing the computationally intensive work, while the sequential code is responsible for the high-level control-flow and I/O. Figure 1 illustrates a fragment of an evolving Calypso execution. Within a parallel step, Calypso supports CREW (concurrent read, exclusive write) semantics to shared data structures, with updates visible only at the end of the current step. Additionally, the parallel tasks are idempotent, implying that a code segment can be executed multiple times (with possibly some partial executions), with exactly-once semantics. These multiple executions mask any faults in the underlying resources.

Calypso augments standard C++ with four keywords: `shared`, `parbegin`, `parend`, and `routine`. Globally shared variables are declared using the keyword `shared`. `parbegin` and `parend` help delimit a parallel step consisting of a sequence of `routine` statements:

```
parbegin
  routine [int-exp](int width, int number)
     routine-body1
  routine [int-exp](int width, int number)
     routine-body2
parend;
```

The `routine` statements specify the tasks within the parallel step: *routine-body1* and *routine-body2* are sequential C++ program fragments, *int-exp* specifies an integer expression indicating the number of copies of each routine that need to be created within the parallel step, and *width* and *number* are arguments

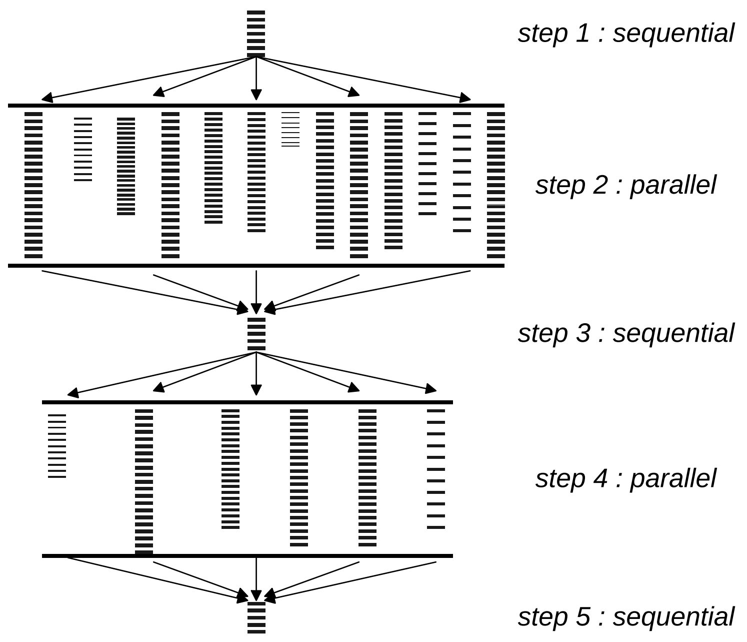

Figure 1: A fragment of an evolving Calypso execution

provided to each task denoting, respectively, the number of tasks created and the sequence number of the specific task among these tasks. As shown in the code fragment above, each parallel step may consist of multiple `routine` statements. Concurrency exists both inside one routine, as well as among multiple routines within the same parallel step.

3 Resource Management Architecture

The MILAN resource management architecture, shown in Figure 2, consists of two major components: an application-level *QoS agent* and the system-level *QoS arbitrator*.[2] The QoS agent communicates the application resource and predictability requirements to the QoS Arbitrator, which satisfies this request (and those from other applications) by providing an appropriate resource allocation. We discuss these components in detail below.

3.1 Components of the Architecture

QoS agent The QoS agent acts on behalf of the application to negotiate an appropriate level of resource reservation/allocation with the QoS arbitrator. The QoS agent, automatically generated from the application's specification by a preprocessing step (see Section 4 for details), describes the application's real-time constraints, its resource requirements, and more importantly its tunability. Tunability is expressed by indicating choice in the execution path available for the application.

As Figure 2 shows, from the perspective of the QoS agent, the application is viewed as an execution path (a chain, or more generally, a dag) comprising several tasks, each with its own resource requirements

[2]The component names signify our focus on providing predictable *quality of service* (QoS) for applications. Here, quality refers both to the desired outputs as well as associated timeliness guarantees.

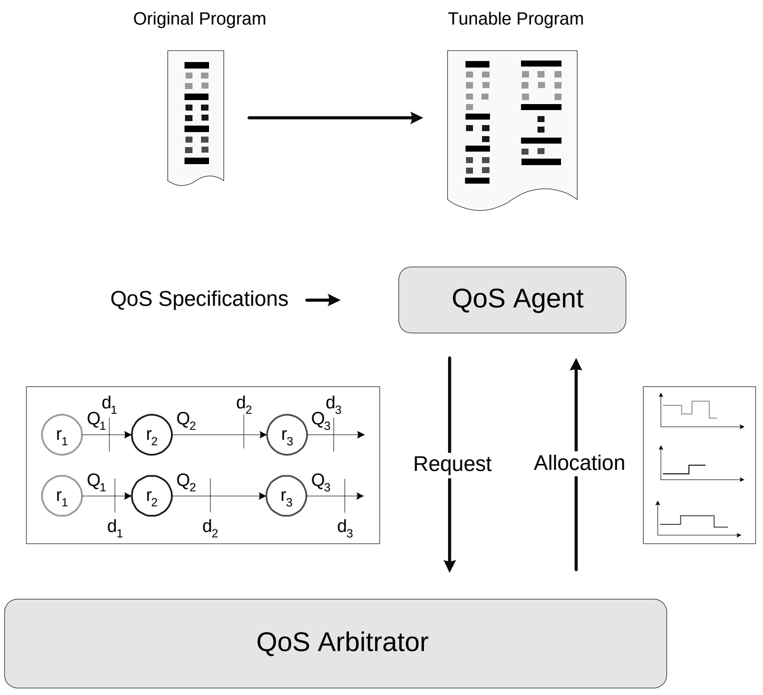

Figure 2: MILAN resource management architecture.

and deadlines. Resource requirements can be thought of as a vector of values, one for each resource in the system. Each task also has an associated output quality, closely related to the requested resources. The quality value of the execution path is obtained by composing the output qualities of each of the tasks. Tunability is expressed by specifying multiple such execution paths, each with its own resource requirement and deadline profiles, representing alternate ways in which the application can consume resources in order to produce outputs with the desired quality.

For the above application model, the Qos agent negotiates a level of resource allocation for each task which maximizes the application output quality. In general, this negotiation can be dynamic possibly involving an initial allocation that gets revised as a function of changing application demands and/or changing system conditions. For the results reported in the rest of the paper however, we restrict our attention to a relatively static negotiation model: the QoS agent communicates all the possible application execution paths and their resource requirements up front, and receives in return (from the QoS arbitrator) a resource allocation profile for one of these paths.

QoS arbitrator The QoS arbitrator takes advantage of the flexible program specification provided by QoS agents to enhance system utilization while satisfying the predictability requirements of each application. In MILAN, this flexibility comes from two aspects. First, application tunability provides the freedom to choose a macro-level resource allocation over time for each application. And second, the adaptiveness of the underlying fault-masking techniques (two-phase idempotent execution and eager scheduling) provide micro-level flexibility, permitting preemptive allocation, deallocation, and reallocation of resources. In this paper, we restrict our focus to the flexibility obtained from application tunability.

Upon job arrival, the QoS arbitrator first performs admission control to check whether or not application resource requirements can be satisfied. Application tunability increases the likelihood that an application can be admitted into the system. The QoS arbitrator scheduling algorithms (discussed in Section 5) first choose the best execution path, and then make an assignment of which processors will execute which application tasks and for what time. These decisions are communicated back to the application QoS agent which configures the application appropriately. In general, the QoS arbitrator also monitors system resources, and triggers renegotiation on detecting a significant change in resource levels (e.g., on a fault, or when new resources become available as in a metacomputing environment). For the purposes of this paper however, we assume that the underlying system is fault-free and has a fixed amount of available resources.

3.2 Junction Detection: An Example Tunable Application

The junction detection [11] application detects distinguished pixels in an image where the intensity or color changes abruptly. Junction detection is a core component of several image-processing applications, often serving as a precursor to shape construction and classification tasks. Our junction detection algorithm consists of three steps. The first step samples a subset of the pixels in parallel and performs a quick test to determine whether or not the tested pixel is of interest. A pixel is of interest if the difference among intensities/colors of its neighbor pixels is beyond a threshold. The second step draws a region of interest around a cluster of interesting pixels. The region is essentially a convex hull containing at least a certain number of interesting pixels in close proximity. Finally, the third step runs a compute-intensive algorithm for every pixel in the regions of interest.

Junction detection is a tunable application because the granularity of sampling in the first step can be parameterized, resulting in different resource requirements. The computation can compensate (with respect of result quality) for a coarser sampling in the first task by possibly drawing additional and/or larger regions of interest. Thus, a smaller resource requirement in the first step (for coarser sampling) is compensated for by requiring a larger resource allocation in the third step. Figure 3 demonstrates this tunability, showing two configurations with different sampling granularities, different thresholds for drawing the regions of interest, and consequently different resource requirements for the third step.

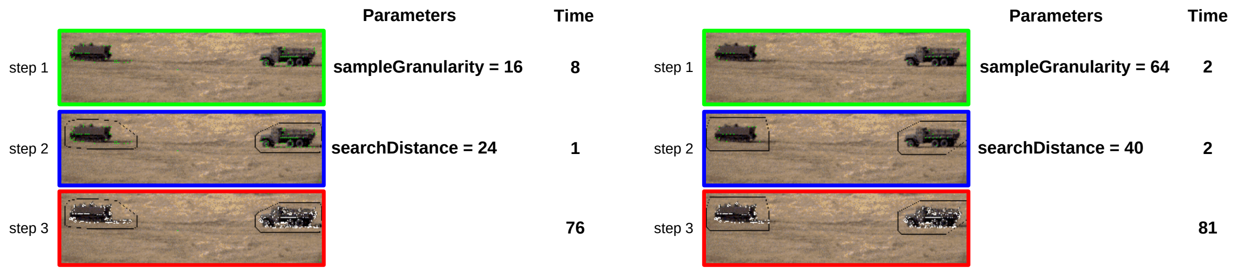

Figure 3: A tunable junction-detection algorithm.

The QoS agent for this application represents tunability in terms of two parameters: the sampling granularity, and a search distance metric which determines how regions of interest are constructed in the second step of the algorithm. The sampling granularity parameter affects the number of pixels sampled in the first step, with coarse sampling being compensated for by a higher value of the search distance (correspondingly, a larger or more regions of interest). The resource requirements, deadlines, and output qualities associated with each alternate execution path are assumed to be available a priori (these can be obtained by profiling on a training set of representative images). The QoS agent communicates this task system to the QoS arbitrator, which chooses the path that will be executed. Note that depending upon system load, different paths may get chosen for junction-detection jobs which arrive at different times. The QoS agent then configures the application to execute along that path. In this case, application configuration just requires setting values for the sampling granularity and search distance parameters.

4 Language Support for Tunability

We extend the Calypso programming language to support expression of tunable, predictable applications. These extensions primarily associate resource requirements and output quality values with individual tasks, specify how tasks may be configured as a function of available resources, and finally specify alternate paths of execution through the program. The Calypso preprocessor uses these extensions to construct a QoS agent for the program which embodies the task graph and tunability aspects of the application. As described in Section 3, during job startup time, this QoS agent negotiates with the QoS arbitrator for an appropriate level of resource allocation.

We first discuss in general the different models of application tunability, describe tunability support in Calypso, and finally examine how the junction detection program introduced in Section 3 can be expressed using these extensions.

4.1 Different Models of Tunability

Tunability modifies the behavior of an application in response to available resources. There are two orthogonal dimensions along which such behavior modification can be expressed: granularity of code modification, and the granularity of resource change which can be detected and acted upon by the program. The first dimension refers to how application behavior modification is effected. Application behavior is controlled both at the macro-level by the algorithms it uses (*coarse tunability*), and at the micro-level by certain parameters that affect control-flow decisions (*fine tunability*). The second dimension refers to an application's ability to respond to a change in available resources. Some applications may be able to take advantage of even a very small change in resources (*continuous tunability*), while other applications can only respond to a finite set of specific resource levels (*discrete tunability*).

Applications typically exhibit tunability corresponding to three of the possible four combinations: coarse-discrete, fine-discrete, and fine-continuous. The first two situations are probably most common where an application can use different algorithms (e.g., compression algorithms in a continuous media application) or modify its control parameters in response to a change in resource availability. However, the application can only react to discrete resource levels, not the entire range. An example of the third situation is the sampling step of our junction-detection program: the sampling granularity serves as a knob which can vary application resource requirements over a continuous range.

4.2 Calypso Extensions for Tunability

These extensions to the Calypso programming language identify program control parameters, alternate execution paths, and the corresponding resource and timeliness requirements of program tasks. To keep our preprocessor simple, we focus on supporting only two models of tunability: coarse-discrete (where tunability is achieved by using different algorithms), and fine-discrete (where tunability is effected by modifying parameter values).[3] Moreover, we restrict our attention to computations where the sequence of parallel steps is independent of program control flow. Most parallel applications satisfy this assumption.

The language extensions can broadly be classified into three categories: *declaration* constructs (for identifying control parameters), *task* constructs (for associating resource and timeliness requirements with tasks), and *structure* constructs (for specifying alternate execution paths in the code). Control parameters are declared (and optionally initialized) within the `task_control-_parameters` block, used as in the following code fragment:

```
task_control_parameters {
   ...
};
```

These parameters are used by the QoS agent, after receiving an allocation of resources from the QoS arbitrator, to appropriately configure the program.

The task construct, `task`, acts as a wrapper around a sequential or parallel Calypso step, and specifies the task deadline, its control parameters, and resource requirements and output quality values for each of the acceptable task configurations. The latter correspond to the assignment of specific values to the control parameters. The following code fragment demonstrates usage of the `task` construct:

```
task name [ deadline]
          [ parameter-list]
          [({ parameter-values},{ resource-request}, quality), ... ]
   // Calypso code for the task
taskend
```

The *deadline* denotes the time within which this task should complete. The *parameter-list* is a list of control parameters which will be assigned a value exactly before this task starts at the execution time. The acceptable task configurations are shown as a list in the last argument of the **task** construct. Each configuration consists of the *parameter-values*, which is a list of value assignments to the control parameters identified in *parameter-list*, the *resource-request*, which is a vector of values of size equal to the number of resource types, and the *quality* which describes the quality value of the task output for the current configuration. For the purposes of this paper, *resource-request* is a processor-time tuple, denoting the number of processors required for the task and the time duration they are required for.

Two structuring constructs are provided to enable selection of an execution path through the program. The `task_select` and `task_loop` constructs are used to represent choice of configurations within a parallel step, and overall iterative structure of the program, respectively. The following code fragment demonstrates their use:

```
task_select
   when  when-expr
      task, task_select, or task_loop constructs
   finally  finally-code
   ...
   when  when-expr
      task, task_select, or task_loop constructs
   finally  finally-code
task_selectend;

task_loop ( loop-expr)
```

[3]Supporting fine-continuous tunability requires that the preprocessor be able to handle symbolic expressions for resource requirements and deadlines. This is more an implementation limitation rather than a fundamental issue.

```
    task, task_select, or task_loop constructs
task_loopend;
```

task_select permits selection among multiple tasks with the same step whose readiness is checked using the *when-expr*. The *finally-code* is executed upon completion of the task and together with the **when** construct permits execution paths to be defined in the program. The **task_loop** construct is used to express an iterative structure surrounding the sequential and parallel steps. Its argument, *loop-expr*, denotes the number of iterations. Both *when-expr* and *loop-expr* can only include constants and control parameters, facilitating their evaluation at scheduling time.

The Calypso language extensions are orthogonal to the parallelism constructs, enabling tunability to be incrementally incorporated into an application program.

4.3 Expressing Tunability in the Junction Detection Program

We next demonstrate the use of the above language constructs in the context of the junction detection program described in Section 3. As mentioned earlier, the program is tunable with respect to two parameters, controlling the sampling granularity in the first step, and the search distance in the second step. It is assumed that the resource requirements and deadlines for each step are obtained by separately profiling the program. Here we focus on how application flexibility and these predictability requirements can be expressed.

Figure 4 shows the pseudo code for the original (fixed) and tunable versions of the junction detection program. The sampling granularity and search distance parameters are identified as control parameters whose values will be provided by the QoS agent based upon its negotiation with the QoS arbitrator. The rest of the program uses these parameters to control the application behavior. The tunability in the first step, `sampleImage`, is expressed using the **task** construct: the arguments state that the deadline for the step is `10.0` units, the step behavior is controlled by the `sampleGranularity` parameter, and allowable configurations correspond to `sampleGranularity=16` and `sampleGranularity=64`. The first configuration requires 4 processors for 8.0 units of time, while the second requires 4 processors for 2.0 units of time.

The second step, `markRegion`, admits coarse-discrete tunability, allowing use of different algorithms based on the sampling granularity used in the first step. As the code fragment shows, this level of tunability is conveniently expressed using the `task_select` construct. The `finally` code fragments are used to set up another control parameter, `c`, which is used to describe allowable task configurations in the third step. Based on the value of `c` (i.e., the task configuration chosen in the second step), only one of the `computeJunctions` configurations is allowed. This restriction of configurations based on which configurations were selected in an earlier step make explicit the application's ability to tradeoff resource requirements over its lifetime. A lower allocation of resources in the sampling step require a larger allocation of resources in the junction computation step.

5 Performance Impact of Tunability

To understand the performance impact of tunability, we first propose a greedy heuristic for scheduling tunable parallel applications with predictability requirements, and then using a synthetic task system systematically quantify the benefits and shortcomings of tunability. The QoS arbitrator component of the resource management architecture described in Section 3 will incorporate this heuristic.

ORIGINAL PROGRAM

```
for ( i=0; i<1000; i++ ) {

   //  routine for sampling the image

   //  algorithm A for finding region of interest

   //  routine for computing junctions

}
```

TUNABLE PROGRAM

```
task_control_parameters {
   int sampleGranularity;
   int searchDistance;
   int c;
};

task_loop( 1000 )
for ( i=0; i<1000; i++ ) {

   task sampleImage[10.0][sampleGranularity][({16},{4,8.0},1.0), ({64},{4,2.0},1.0)]
     //  routine for sampling the image
   taskend;

  task_select
    when (sampleGranularity==16)  task markRegionA[60.0][searchDistance][({24},{2,1.0},1.0)]
                                   //  algorithm A for finding region of interest
                                  taskend;  finally {c = 0;}
    when (sampleGranularity==64)  task markRegionB[60.0][searchDistance][({40},{2,1.0},1.0)]
                                   //  algorithm B for finding region of interest
                                  taskend;  finally {c = 1;}
  task_selectend;

  task_select
    when (c==0)  task computeJunctions[100.0][][({},{4,76.0},1.0)]
                  //  routine for computing junctions
                 taskend;  finally {}
    when (c==1)  task computeJunctions[100.0][][({},{6,81.0},1.0)]
                  //  routine for computing junctions
                 taskend;  finally {}
  task_selectend;

}
task_loopend;
```

Figure 4: Expression of application tunability in the Junction Detection program.

5.1 Scheduling Formulation

Without tunability, the underlying scheduling problem we address is one of dynamically scheduling parallel real-time jobs in a system with a fixed amount of homogeneous processing resources. As described in Sections 3 and 4, we restrict our attention to jobs which can be represented as a chain of tasks. Each task is assumed to be non-preemptible, and have a fixed resource requirement "shape" (in terms of the number of processors required over a time period). The non-malleability assumption is true of most current-day parallel applications written in a message-passing style using systems such as PVM [7] or MPI [9]: these applications can execute on a range of machine sizes, but require that the number of processors assigned to the application remain unchanged over the entire lifetime. We also assume that information about all tasks of the job is available upon job arrival. The primary objective of this scheduling problem is to maximize the number of on-time jobs. A secondary objective is to maximize system utilization.

With tunability, the only change to the scheduling problem is that a job is now represented by several task graphs, each of which is itself an OR task graph. These multiple task graphs represent the various alternate execution paths through a tunable program. For uniformity, we assume that all paths through an OR graph have been enumerated, so a tunable application is represented by multiple task chains. So as to not bias one chain in favor of another, for the purposes of this paper, we assume that each chain requires the same total amount of resources and achieves the same output quality value. Note that in practice, task chains of a tunable application are likely to have different overall resource requirements and output qualities: the issue then is of maximizing the achieved job quality.

As with most non-trivial scheduling problems both the above formulations are NP-hard. Furthermore, unlike the uniprocessor case, where there are well-established algorithms for real-time scheduling (such as rate monotonic (RM) [12, 16], earliest deadline first (EDF), and least slack time first (LST) that are all optimal in the sense that if any algorithm can schedule a given task set with no missed deadlines, then all of them can as well), no such algorithms are known for the parallel case. Consequently, in the next section, we describe a greedy heuristic for the above scheduling problem.

5.2 A Greedy Heuristic

The intuition behind why application tunability may improve system performance is that given non-malleable, non-preemptive parallel tasks, the task "shape" determines which tasks can "fit together" in the system. A tunable job is more flexible because it has more than one shape. Our heuristic aims at exploiting this flexibility to achieve better system utilization and throughput.

The heuristic greedily allocates resources to jobs. For a tunable job with multiple schedulable configurations, the heuristic finds among all of them the one that most efficiently uses the system. The heuristic keeps track of available *maximal holes* in the processor-time 2D space: each hole is represented by a triple (t_b, t_e, m) (denoting that m processors are available from beginning time t_b until the end time t_e), and is maximal if it is not contained within any other hole. A job is schedulable if all the tasks on its task chain (any one of the task chains for a tunable job) can be scheduled into available holes while meeting the task deadlines. Ties between schedulable configurations are broken in favor of chains which maximize system utilization (over a time window defined by the job's release time and scheduled finish time) and require fewer total resources for some prefix of their tasks. Under the assumptions of our task model, the heuristic finds the job configuration which achieves the earliest finish time.

Figure 5 demonstrates the working of the heuristic using an example task system. The task system consists of a single tunable job which admits two configurations (*s1* and *s2*), each with two tasks. The task resource requirements are shown in the Figure as *(processor=p, time=t, deadline=d)* triples, signifying that

CONFIGURATION	Task 1			Task 2		
	processor	time	deadline	processor	time	deadline
s1	5	10	30	2	15	60
s2	3	15	30	7	5	60

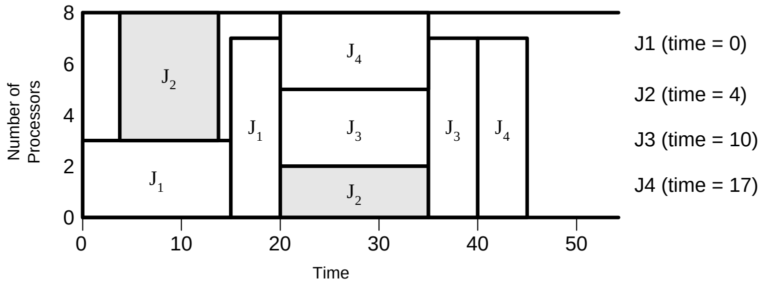

Figure 5: An example task system showing operation of the scheduling heuristic. The holes in the system after each of the jobs J_1, J_2, J_3 and J_4 have been scheduled are shown as a list of (t_b, t_e, m) triples.

the task requires *p* processors for time *t* and must complete before deadline *d*. The figure shows how four jobs arriving at times 0, 4, 10, and 17 are inserted into the system. The sets of triples following each job's name show the holes in the system after each job has been scheduled. Job J_1 starts off with the entire system being available and is scheduled to finish at time 20: configuration *s2* is selected on the basis of its earlier finish time. The list of system holes is modified to indicate the resources committed for J_1. Job J_2 arrives at time 4 and although both configurations are schedulable, *s1* yields an earlier finish time. Job J_3 arrives at time 10 and can be scheduled to finish at time 40: as with J_1, configuration *s2* yields the better finish time. Finally, J_4 arrives at time 17 and the heuristic finds that only one of the configurations, *s2*, is schedulable: the other configuration would end up missing the job deadline because there no available holes before the deadline where it could be inserted. Note that if *s1* were the only configuration (as in a non-tunable job), the job would end up missing its deadline.

5.3 Performance Benefits of Tunability

We use the above heuristic to characterize the performance benefits from tunability. In order to systematically explore the space of application behaviors, we consider a synthetic task system which consists of a parameterizable tunable job shown in Figure 6. The job parameters enable the convenient simulation of a range of job shapes and deadline characteristics.

The parameterizable job consists of two chains, each with two tasks. The two configurations simply transpose the positions of the two tasks. Each task requires the same total amount of resources but with different shapes. One task asks for x processors for time t, whereas the other task requests $x\alpha$ processors for time t/α amount of time. The value of α is chosen in the interval $(0, 1]$ such that both x and $x\alpha$ are integers. Modifying x and α allows the simulation of a variety of task shapes. The task deadlines are set in terms of another parameter, the *laxity* of the job, expressed as the ratio of the slack time in the time period from the release time to the deadline. Since a task can begin execution as soon as its immediate predecessor completes, the task deadline denotes the time by which the task and all its predecessors must finish. The laxity parameter allows the systematic modeling of different amounts of slack time, and hence

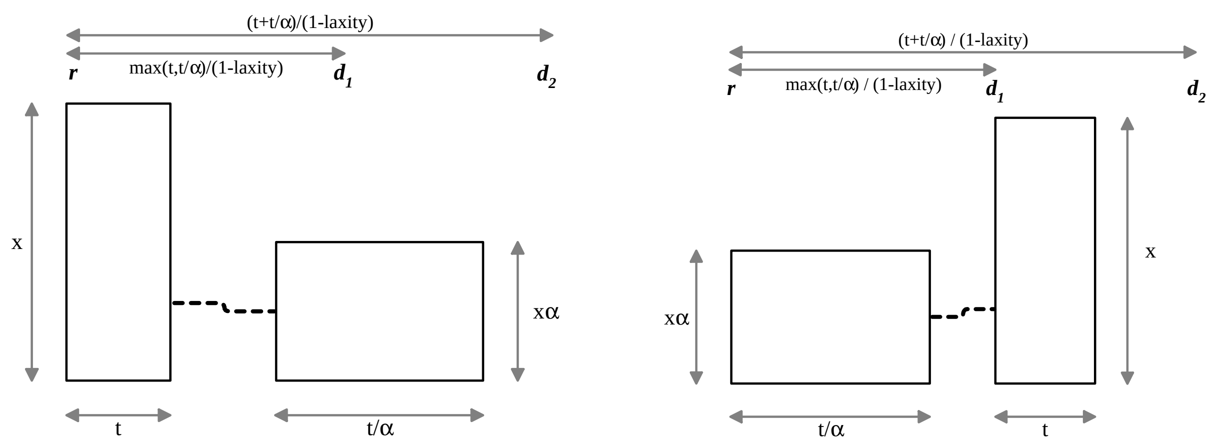

Figure 6: A parameterizable tunable job. The job parameters x, α, and laxity, permit the convenient simulation of a range of job shapes and deadline characteristics.

the constraints that exist for "fitting" a job into the system.

With the above parameterizable job, we construct three task systems: the first task system is tunable, consisting of both job configurations, while the other two systems are non-tunable, containing one configuration apiece. Jobs in each task system are assumed to arrive according the Poisson distribution. We quantify the benefits of tunability in terms of two metrics—*system utilization* and *job throughput*—measuring the performance of the tunable task system as compared to the non-tunable task systems as a function of four parameters: *mean arrival interval*, *laxity*, the *number of processors* in the system, and α which controls the job shape. Figures 7-10 show the system utilization and throughput as these parameters are varied one at a time, keeping the others fixed. All experiments reported in this section assume $x = 8$, $t = 5$, and 10,000 job arrivals. Figures 7-8 show two sets of graphs: these correspond to the (left) absolute and (right) relative (with respect to the non-tunable task systems) throughput and system utilization.

Sensitivity to inter-arrival time In Figure 7, the arrival interval varies from 1 to 20 (note that $t = 5$). When the arrival interval is small, the system is overloaded and only a small portion of the tasks are admitted in all three systems. Tunability has negligible performance impact, since the system is already being 100% utilized. When the arrival interval is very high, the system is overloaded and all three task systems can admit all the jobs. Tunability again does not yield much benefit since resources are abundant compared to requests. It is in the middle range of arrival intervals however, that the tunable system achieves the largest improvement in both utilization and throughput: at its peak, it can admit up to 2000 more jobs and achieve up to 30% better system utilization. Even under moderate overload, a non-tunable system is unable to utilize system resources as efficiently as a tunable system.

Sensitivity to laxity In Figure 8, the laxity varies from 0.05 (a slack time of 5%) to 0.95 (a slack time of 20 times the processing time). As with arrival time, the tunable system yields negligible improvement at the two extremes of the parameter range: when laxity is small, the job deadlines are so tight that there is little space left to tune resource requirements, and when laxity is large, all the systems pack the jobs equally well. Of the two shapes, shape 2 packs really well when deadlines are loose, so performance improvements follow a bell-curve peaking at a laxity of 0.4. In contrast, shape 1 requires a larger number of processors

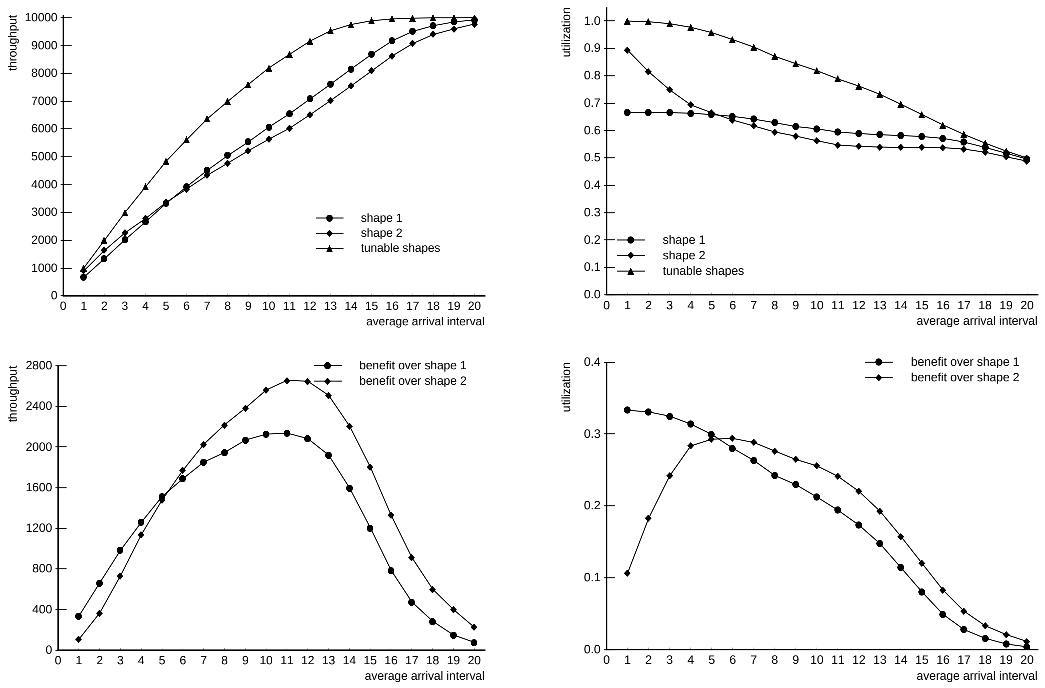

Figure 7: Performance impact of tunability as job inter-arrival time is varied. The left graphs show absolute utilization and throughput, while the right graphs show improvements relative to non-tunable task systems.

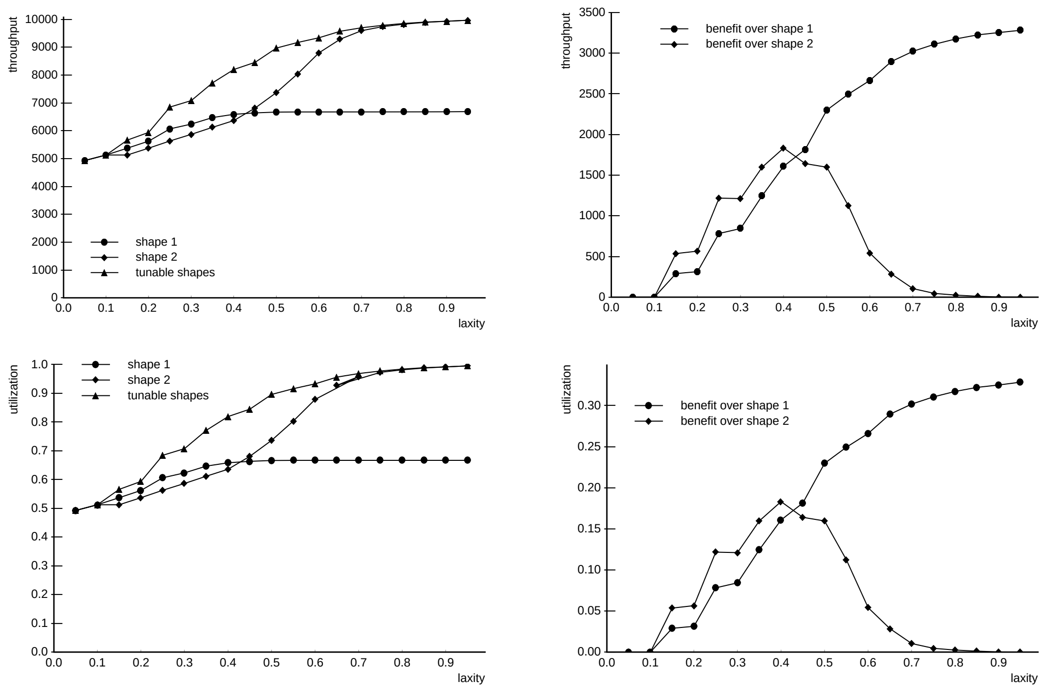

Figure 8: Performance impact of tunability as job laxity is varied. The left graphs show absolute utilization and throughput, while the right graphs show improvements relative to non-tunable task systems.

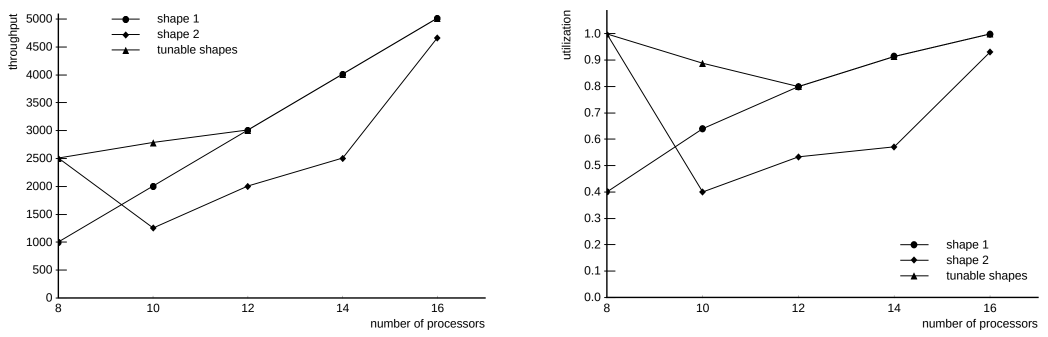

Figure 9: Performance impact of tunability as number of processors in the system is varied.

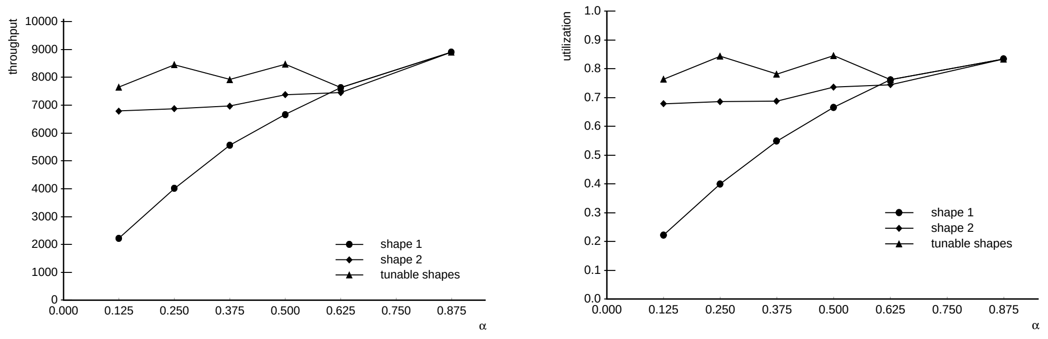

Figure 10: Performance impact of tunability as job shape is varied.

for its first task, preventing its packing (due to the greedy nature of the heuristic) even when deadlines are loose. In this case, system utilization and throughput never exceed 60%, and the tunable task system achieves improvements linearly proportional to the laxity. The latter situation demonstrates the performance handicap of an inflexible (i.e., non-tunable) application.

Sensitivity to the number of processors Figure 9 shows the benefits from tunability when the number of processors in the system are increased from 8 to 14 (recall that $x = 8$). In general, when task concurrency is fixed, more processors would seem to imply increased scheduling flexibility for non-tunable systems and consequently reduced benefit from tunability. The graphs show this trend, especially with respect to the task system consisting only of shape 1 jobs. The graphs show another interesting shortcoming of non-tunable applications: the task system with shape 2 jobs exhibits a parallel anomaly, actually lowering job throughput on 10, 12, and 14 processors as compared to 8 processors. In contrast, the tunable task system exhibits no such errant behavior.

Sensitivity to the job shape Figure 10 shows the benefits of tunability as a function of the job shape, determined by α. Tunability does improve performance but without any discernible trend. The relationship between α and achieved performance is not straightforward because α affects both how well two shapes pack in space, as well as the amount of unutilized space that can be exploited using application tunability. A closer analysis of the results bears out this trend: the situations where performance improvements between the tunable and non-tunable systems are small appear only in border cases where the latter systems benefit from good fortuitous packing. The important issue here is that a tunable job specification provides more robustness in the situations where good packing does not happen, which is likely to be the case in practice.

5.4 Summary

Our simulation results show that tunable task systems yield substantial performance advantages, with respect to both system utilization and job throughput. These improvements are most pronounced in mid-portion of each parameter value range: when the system is moderately loaded, when the deadlines are neither extremely tight nor extremely loose, when the number of processors available is not significantly larger than the concurrency degree of a parallel task, and when the job shapes are not too similar. The above describes the operating point of most practical parallel systems, attesting to the significant performance potential of application tunability.

6 Discussion and Related Work

In this paper, we have examined how to efficiently support applications with predictability requirements in a parallel system environment. The ability to efficiently support such applications ensures that general-purpose media-processing applications, which are likely to be one of the largest application domains in the coming years, can benefit from parallel execution. Our solution, centered around the notion of application tunability, captures the flexibility such applications exhibit with respect to resource requirements. Tunability yields benefits both for the system and the application. While we have demonstrated it in the specific context of predictable parallel computations, the idea of increased flexibility in application specification is relevant in most resource management scenarios; the flexibility can be viewed as increasing an application's likelihood of obtaining good performance with a fixed set of resources.

Predictability for applications has been well studied in the real-time systems literature. Several well-known scheduling schemes, such as rate-monotonic [12, 16] and earliest-deadline-first (EDF), exist for scheduling real-time tasks in an uniprocessor environment. Gillies [8] has studied the scheduling of tasks in AND/OR graphs, a problem very similar to our problem of scheduling task chains in tunable applications. However, the validity of most of these results is restricted to either uniprocessor systems or a sequential task model. Our interest is in managing parallel real-time tasks in a parallel system: in this scenario, well-known results such as EDF no longer provide the optimality guarantees they do in sequential systems.

As described earlier, the primary focus of almost all parallel resource management systems is on optimizing system utilization, often in the context of relatively static resource requirements. The DRMS (Distributed Resource Management System) [14] is an exception, providing system support and resource management for dynamic reconfiguration of parallel programs (with respect to the number of tasks and mapping of tasks to processors). This reconfiguration can be thought of as being closely related to our notion of application tunability; however, unlike the focus of this paper, DRMS exploits this reconfigurability primarily to optimize system utilization and providing only best-effort guarantees to the application.

The Illinois EPIQ project [13, 10] is closely related to our work in that it looks into quality aspects of an application and tries to trade off output quality for resource requirements. The EPIQ view is that the quality of an execution degrades when there are insufficient resources. Our work stresses the tunable aspects of applications, especially the ability to tradeoff resources over time with the same QoS goal. The EPIQ work is also different in focusing on a general framework for specifying and negotiating QoS values. In contrast, we adopt a simpler model of resource versus quality tradeoff to focus on algorithmic issues surrounding the use of these tradeoffs to improve system performance.

7 Conclusion

We have presented a novel approach for improving parallel system utilization while simultaneously meeting the predictability requirements of general-purpose parallel applications such as image recognition, media processing, and virtual reality. Our approach takes advantage of *application tunability*, which refers to a flexibility in the application specification particularly with respect to its ability to trade off resource requirements over time, while maintaining a desired level of output quality. A resource management system can exploit this flexibility to decide how best to allocate resources to predictable computations so as to maximize both job throughput and overall system utilization. We have described language extensions to the Calypso parallel programming language for expressing tunability, and evaluated its performance impact using a synthetic task system to systematically identify its benefits and shortcomings. Our results show that application tunability is convenient to express, and can yield significant performance benefits.

Current work is focused on implementing and evaluating support for application tunability in the MILAN parallel computing environment using large-scale image processing applications.

8 Acknowledgment

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In Proc. 4th IEEE International Symposium on High Performance Distributed Computing, 1995.

[2] A. Baratloo, M. Karaul, Z. Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. In Proc. of 9th International Conference on Parallel and Distributed Computing Systems, September 1996.

[3] K. Chandy and C. Kesselman. A description of CC++. Tech. Rep., CS-92-01, California Institute of Technology, 1992.

[4] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. in 9th Joint Symposium on Parallel Processing (JSPP'97), Kobe, Japan .

[5] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: A fault-tolerant high performance approach. in Proc. 15th IEEE International Conference on Distributed Computing Systems, 1995.

[6] K. Diefendorff and P. Dubey. How Multimedia Workloads Will Change Processor Design. IEEE Computer, Vol. 30, No. 9, Sep. 1997

[7] G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. Concurrency:Practice and experience, 4(4):293-311, 1992.

[8] D. Gillies and J. Liu. Scheduling Tasks with AND/OR Precedence Constraints. Proc. of 2nd IEEE Conference on Parallel and Distributed Processing, Dec. 1990.

[9] W. Gropp, E. Lusk, A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. MIT Press, 1994, ISBN 0-262-57104-8.

[10] D. Hull, A. Shankar, K. Nahrstedt, and J. Liu. An end-to-end QoS model and management architecture. In Proc. of IEEE Workshop on Middleware for Distributed Real-time Systems and Services, California, Dec. 1997.

[11] H. Ishikawa and D. Geiger. Segmentation by Grouping Junctions. in Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'98), 1998.

[12] C. Liu and J Layland. Scheduling Alorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of ACM, Vol. 20. no. 1 Jan 1973.

[13] J. Liu, K. Nahrstedt, D. Hull, S. Chen, and B. Li. EPIQ QoS characterization. Draft, July 1997. http://pertsserver.cs.uiuc.edu/epiq.

[14] J. Moreira and V. Naik. Dynamic Resource Management on Distributed Systems using reconfigurable Applications. IBM J. of Research & Development, vol. 41(3), 1997.

[15] S. Sardesai and P. Dasgupta. Chime : A Versatile Distributed Parallel Processing Environment. http://cs.nyu.edu/milan/milan/index.html.

[16] L. Sha and J. Lehoczky. Performance of real-time bus scheduling algorithms. ACM Perform. Eval. Rev., 1986, 14(1).

[17] The MILAN Project: Metacomputing in Large Asynchronous Networks. http://www.cs.nyu.edu/milan/milan/index.html.