Rate-Control as a Language Construct for Parallel and Distributed Programming

Peter H. Mills, Jan F. Prins[†], John H. Reif

Department of Computer Science,
Duke University,
Durham, N.C. 27708-0129

[†]Department of Computer Science,
University of North Carolina,
Chapel Hill, N.C. 27599-3175 USA

Abstract

This paper introduces a new parallel programming language construct, the *rate* construct, and examines its utility for a variety of problems. The rate construct specifies constraints on the relative rates of progress of tasks executing in parallel, where *progress* is the amount of computational work as measured by elapsed "ticks" on a local logical clock. By prescribing expected work, the rate construct constrains the allocation of processor-time to tasks needed to achieve that work; in a parallel setting this constrains the distribution of tasks to processors and multiprocessing ratios, effected for example by load balancing. We present definitions of rate and underlying real-time primitives as orthogonal extensions to the architecture-independent parallel programming language *Proteus*. The utility of the rate construct is evidenced for a variety of problems, including weighted parallel search for a goal, adaptive many-body simulation in which rates abstract the requirements for load-balancing, and variable time-stepped computations in which the use of rates can alter the frequency of asynchronous iterations.

1. Introduction

High-level parallel programming languages, and the simple computing models they reflect, are essential tools in exploring the behavior of parallel algorithms at a suitable level of abstraction. By abstracting away implementation details, a careful choice of high-level parallel language concepts can provide descriptive simplicity and promote machine-independence critically needed to facilitate the task of parallel programming. However, several concerns arise in the high-level approach to the expression of parallel algorithms. First, there is often wide gap between the idealized programming models presented by parallel languages and the diverse low-level machine-specific languages in which an implementation is to be realized. Much attention has turned towards program transformation as a natural approach to bridge this gap, for example in the efforts of the Crystal equational language [CCL91], variants of Bird-Meertens functional formalism [Ski90], the NESL nested data-parallel language [Ble92], and *Proteus* [MNPR92, PP93]. The gap between abstraction and practical realization also impacts whether properties such as performance can be preserved under transformation, an open problem.

Secondly, and in a related vein, the question arises of what exactly should be abstracted and in what form. Often the idealized programming models and languages used to specify parallel algorithms do not adequately reflect key characteristics of the practical realizations, or do so at an inappropriate level of detail – that is, they can not express well if at all certain concepts which impact correctness or performance. And so it is important to strike a careful balance between being sufficiently abstract to be simple and machine-independent, yet detailed enough to reflect realistic properties such as performance [Ski91]. For example, considerations of data locality are difficult yet desirable to address abstractly since they can strongly affect performance. In this paper we consider another domain, that of computational progress, and examine the utility of a control abstraction which constrains the relative rates of progress of tasks executing in parallel. Specifying the relative rates of computational progress is in essence a novel abstraction of real-time requirements.

Current real-time features of programming languages do not address notions of computational progress at a high level of abstraction, if at all. Real-time systems are concerned with the temporal correctness of systems – in addition to the functional correctness of systems – in that the system must satisfy timing requirements such as deadlines for response or delays before acting. The method for satisfying these constraints is either by statically determining the feasibility of scheduling the required work on available resources in a way that meets the timing constraints, or by dynamically issuing scheduling directives. However, conventional timing constraints are typically too low level to effectively specify progress in a parallel setting. Some notion of progress is found in language features such as "engines" in MultiLisp and Scheme [Dyb87]. Engines typically employ timeouts to execute a function for a certain number of steps (fuel) before returning a continuation. However, engines constrain progress at a low level of detail and are really useful only in a sequential or multiprocessing situation.

Yet the expression of progress constraints is a valuable abstraction. By succinctly specifying expected work, progress constraints abstract the allocation of

[‡]This work was supported under DARPA / SISTO contracts N00014-88-K-0458, N00014-91-J-1985, N00014-91-C-0114 administered through ONR, NASA subcontract 550-63 of prime contract NAS5-30428, US-Israel Binational NSF Grant 88-00282/2, and NSF Grant NSF-IRI-91-00681.

First IEEE Workshop on Parallel and Distributed Real-Time Systems,
(IPPS'93, Newport Beach, California, April 13-16), 1993.

computational resources required for that distribution of work. An advantage of specifying rates of progress thus lies in that it translates what might otherwise be timing constraints into resource requirements, a strategy which can potentially decrease run-time overhead that would be spent in satisfying lower-level timing constraints, or reduce the difficulty of compile-time checks for timing feasibility. A notion of rates saddles at a high level both explicit resource management – scheduling and mapping directives to distribute work load – and real-time primitives which determine allocation of resources in order to meet low-level timing constraints. Furthermore, conveying progress information can succinctly specify aspects of functional correctness, and can serve to give detail which might aid program transformation and support more rigorous performance analysis.

In this paper we introduce the *rate* construct, and examine its utility through a variety of problems. The rate construct specifies constraints on the relative rates of progress of tasks executing in parallel, where *progress* is the amount of computational work measured by elapsed "ticks" on a local logical clock. By prescribing expected work, rates in turn constrain the allocation of processor-time to tasks needed to achieve that distribution of work over time. In a single processor setting, rates can thus dictate the relative percentage of time to receive in multiprocessing. In a parallel setting, rates can be interpreted as determining the mapping of tasks to processors and multiprocessing ratios, effected for example by load balancing. The rate construct thus attempts to fill a gap in the expression of resource requirements such as computational progress which real-time constructs do not adequately address. The succinct expression of such requirements is needed to explore parallel algorithms at a suitable level of abstraction. We envision that the rate construct, in conjunction with abstractions for specifying data locality still under development, might be used under transformation to generate lower-level mapping and scheduling directives.

The remainder of this paper begins in Section 2 with a brief overview of the architecture-independent parallel programming language, *Proteus*, which provides a convenient framework for expressing the rate construct. In Section 3 definitions of relative rate and a simpler notion *fixed-rate*, which reduces relative rates to fixed rates of absolute progress with few assumptions of global synchronization among processors, are given in terms of underlying notions of time and clocks. In Section 4 the utility of the rate construct is evidenced for a broad span of applications, including:

- *minimizing parallel work*, for example in weighted parallel search for a goal, where several search paths are pursued each having different likelihoods of success.
- *altering frequency of asynchronous iteration*, for example by using fictitious rates in variable time-stepped computations.
- *abstracting requirements for load-balancing*, for example in adaptive solutions to irregular problems such as many-body simulation.

In Section 5 a more complete set of real-time primitives is presented for *Proteus*, and in Section 6 further mechanisms for resource allocation are examined. We conclude the paper with a discussion of ongoing research.

2. The *Proteus* language

The rate construct and underlying real-time primitives, while in essence language independent, are succinctly presented in this paper as extensions to *Proteus*. *Proteus* is a high-level architecture-independent parallel programming language being developed to support the prototyping of parallel algorithms [MNP+91, MNPR92]. The goal of the *Proteus* effort is to overcome the difficulties of expressing and evaluating design alternatives for parallel applications posed by the tedium of programming in non-portable low-level machine-specific parallel languages. Our approach is to construct high-level machine-independent programs using *Proteus*, and rely on transformation techniques to realize implementations on diverse parallel architectures. The *Proteus* language specifies parallelism at a high-level using succinct notations based on sets and sequences, and allows program behavior to be validated by sequential simulation. Actual parallel execution can be achieved by program transformations which place it in a restricted form that can be translated directly to a low-level machine-specific parallel language. The source-to-source transformations and translation can be managed semi-automatically by the analysis and theorem-proving capabilities of KIDS (Kestrel Interactive Development System) [Ref88, Smi90].

For example, we have investigated refinement strategies that transform a broad class of high-level data parallel operations over sequences into the widely portable vector language CVL. The language and transformation techniques are also being used to prototype sophisticated parallel algorithms for many-body interaction as used in molecular dynamics simulations [MNPR92].

We describe here only a few salient features of *Proteus*. The *Proteus* language employs as its principle data types the high-level mathematical notions of sets, sequences, and maps, in a manner similar to such languages as SETL [SDDS86] and REFINE [Ref88]. Sequence (similarly set) comprehension refers to the construction of a sequence by generation based on another sequence, of the form:

$$[\text{expr(x)} : \text{x } \mathbf{in}\ S \mid \text{pred(x)}]$$

Sequences can also be constructed by enumeration.

Proteus provides a succinct yet powerful set of primitives for the parallel composition and synchronization of processes communicating through shared memory. Parallel execution is explicitly specified with a simple parallel composition operator, of the form:

$$(S_1 \| \ldots \| S_n) \qquad or \qquad \mathbf{parfor}\ i\ \mathbf{in}\ [1..n]\ (S_i)$$

This specifies "**cobegin/coend**"-like concurrent execution of the statements S_i, which are considered to be processes, with no assumptions about atomicity, interleaving, or progress. It can also be specified in iterated form, as shown on the right above. The processes of

this construct must terminate before the construct can complete.

Proteus provides a few simple mechanisms to control interference and support synchronization. One mechanism is the provision of declarations which partition the global state – that is, variables that are nonlocal to a process using standard lexical scoping – into *private* and *shared* variables. Parallel processes operate on individual copies of private variables which may be merged into the shared state at specifiable barrier synchronization points. This mimics the PRAM model of computation often used to design parallel algorithms. Private variables and other features of *Proteus* are more fully described in [MNP+91, MNPR92].

3. Progress constraints

We now describe mechanisms for specifying time and progress-constrained computation, and illustrate their utility on a variety of problems. While it is critical to be able to specify and analyze the real-time behavior of programs, it is also advantageous, particularly in prototyping, to extend the management and analysis of resource requirements at an abstract level to other domains such as computational progress. To this end we introduce the **rate** construct, an augmentation to the parallel construct which models the relative rates of progress of the parallel processes. We now consider what such a construct means, both informally and formally.

3.1. The rate construct

Intuitively, the **rate** specification indicates, for each process, the proportion of total execution steps to take for each period of a common virtual clock. For example,

$$(P_1 \parallel P_2 \parallel P_3)\ \textbf{rate}\ [0.2,\ 0.2,\ 0.6]$$

specifies that P_3 takes 3 steps (according to the ticks of some virtual clock) for every step of P_1 and P_2. [4] While in this example we informally assume the rate constraint must be obeyed exactly, we later weaken such a restriction to make it more practical to realize. The informal intuition of the rate construct prompts several questions. How do we measure virtual time – that is, when the clock "ticks"? What constitutes a "step" of the computation – that is, what are the means of correlating virtual ticks to execution times of statements in the processes, so that elapsed time on the clock describes computational progress? Perhaps most importantly, how can a common virtual clock be realized for a group of asynchronous processes – what overhead for global synchronization is required to maintain a notion of global time? Such global information must be present or implied in some form since the rates are *relative* to the slowest process.

These questions can be answered by making the underlying notions of clocks and steps more precise. Following Lamport [Lam78], we view a process as a set of *events*, each consisting of the occurrence of some action or computation step (not necessarily atomic). These events are partially ordered in that one event may cause another. [5] With each process P_i we can associate a *clock*, which is an abstraction that is nothing more than a non-decreasing function which assigns a number $C_i\langle a\rangle$ to each event a in P_i. Since no assumption is made about the relation of C_i to physical time, these are called *logical clocks* (or virtual clocks). For our purposes we wish each C_i to measure the amount of *logical work* done so far, so we increment each C_i only when it completes a local action.

What constitutes an action, and how much do we increment C_i? One simple measurement technique, valuable for prototyping, is to let the clock be updated in user-specified increments by user-designated process events. That is, expected durations can be explicitly attached to statement (or "timing") blocks. *Proteus* provides such a timing specification through the construct,

$$(\textbf{duration}\ \alpha)\ (S)$$

which specifies the duration of the event constituting the execution of statement block S. That is, after the completion of the statement S in P_i, the clock C_i is incremented by α. In a similar manner, a timing constraint for delay,

$$\textbf{wait}\ \alpha$$

if executed defines an event of duration α which updates the clock for that process. The **duration** construct is similar in form to that of the Flex language, in which conditions specifying inequalities on start, finish, interval and duration times can be attached to timing blocks [KL91].

The **rate** construct can now be defined, more precisely, as specifying a constraint on the relative rates of progress of tasks executing in parallel, where *progress* is the amount of computational work as measured by elapsed "ticks" on logical clocks local to each process, updated in user-specified increments by process events. For the moment, let us defer issues of global synchronization, and introduce a global notion of continuous time, t, which we as external observers can perceive. Let $C_i(t)$ denote the value of clock C_i at "physical" time t. It must be emphasized that the C_i still measure logical work, and are not to be regarded as physical clocks in the sense of Lamport [Lam78]. Then the rate construct implies the constraint:

Rate condition: $\forall P_i, P_j \in P$,

$\forall t$: P_i nor P_j have terminated,

$$|C_i(t)\frac{rate_j}{rate_i} - C_j(t)| < \tau$$

where τ is an bound on "slippage" that can be explicitly specified in *Proteus* via the construct

$$\textbf{interval}\ \tau$$

τ places a bound on how accurately the ratios of computational progress must obey the rate directive, and is to be given in the same unit of measure used in the

[4] If the rates do not sum to 1 they are normalized to do so.

[5] In the case of a sequential process the temporal ordering of these events is total; for distributed systems these events may be partially ordered by a causality relation.

duration construct. For example, letting $(S)*$ denote repeated iteration of statement S,

$$(((\textbf{duration } 1)\ P_1)* \parallel ((\textbf{duration } 1)\ P_2)*)$$
$$\textbf{rate } [0.66,\ 0.33]\ \ \textbf{interval } 1$$

means that P_1 will iterate roughly twice as fast as P_2, but that P_1 may get ahead or behind by as much as one iteration. If the **interval** specification is omitted, it is assumed to be some reasonable finite duration that may be implementation-dependent.

Such a "slippage" bound provides some flexibility in implementation, and might be used to yield a derived sampling interval which says how often absolute progress must be consulted and adjusted to maintain the relative rates. This does not guarantee that over any given period we will observe at least once an exact admissible ratio of computational progress, but it does mean the approximation is close.

There are still two unresolved issues in this formalization. The first concerns the fact that there is still a notion of global ("physical") time, and secondly that a process P_i must have knowledge of the progress of P_j. The notion of global time can, without loss of generality, be ignored in several ways. One way is by replacing t with another logical clock on each process which abstracts physical time. It is well known that such physical clocks can be synchronized within a fixed small time interval [Lam78, MO83], and we assume that the granularity of timing constraints and τ are large compared to that error. Another way, roughly equivalent to the first, might be to define a common virtual clock and use a type of rounds number [LF81]. [6] More important, however, is that the general notion of rate encompasses variable-speed processes (resulting from time-varying processor apportionment), so that much global progress adjustment may be necessary if any one process is allowed to slow arbitrarily. This in turn implies a potentially large amount of monitoring and synchronization overhead.

3.2. Fixed rates of absolute progress

To simplify the problem of global synchronization, we make some assumptions which also ease the implementation details. A simpler interpretation of the rate construct, *fixed-rate*, is introduced which runs each process at a fixed absolute rate whose ratios obey the rate constraints. In other words, each process promises to run at a fixed rate of absolute progress which satisfies the relative rate constraints. This essentially apportions an unvarying percentage of logical processor time to each process, and entails no global adjustments of absolute progress among processes. The problem of tracking relative rates between processes is thus reduced to ensuring absolute rates of progress of each process on its processor. This can prove advantageous, for example, in modeling totally asynchronous distributed algorithms such as real-time resource granting systems which make assumptions about relative processor rates [RS82].

[6] By obeying certain local clock update rules and exchanging messages, a common clock consistent with local clocks can be derived [Lam78, LF81].

Formally, for a given starting time t_0 of the parallel construct and an arbitrary constant β, we require that:

Fixed-rate condition: $\forall P_i \in P$,

$$\forall t : t > t_0 \text{ and } P_i \text{ has not terminated},$$
$$|\frac{(t-t_0)rate_i}{\beta} - C_i(t)| < \frac{\tau}{\gamma+1}$$

where $\gamma = (max_{P_j \in P} rate_j)\ /\ (min_{P_j \in P} rate_j)$

It is easily verified that *fixed-rate condition* implies that the *rate condition* will be satisfied.

The choice of β acts as a scaling factor for the rates which determines the percentage of multiprocessing time each process receives; for ease of programming it is assumed the system automatically scales the rates reasonably. β can be chosen, for example, to be roughly the time to take one step:

$$\beta = \frac{1}{\tau} \max_{P_i \in P} (\min_{\alpha \geq 1} (minC_i^{-1}(\alpha+\tau) - maxC_i^{-1}(\alpha)))$$

3.3. Variable rates and nested rates

The rate specification may be reset dynamically by any individual process; in this case all processes in the affected parallel composition must re-adjust their progress. The rate specifications may also be nested: in that case, the expected rates multiply down the process hierarchy and the minimal slippage is chosen. Furthermore, a range of rates may be specified, for example:

$$(P_1 \parallel P_2 \parallel P_3)\ \textbf{rate } [[0.2,\ 0.4], [0.2,\ 0.4], [0.6,\ 0.8]]$$

In this case any combination of relative rates within the ranges which sums to 1 is admissible. A further generalization allows the specification of rates as probability variables – that is, giving rate constraints as a function from rates to expected probability – yielding a probabilistic rate control. The semantics then associates, for each admissible execution history, a probability in a manner analogous to that for variable-speed APRAM's [CZ89].

The **rate** construct proves advantageous in several ways. The specification of expected rate of progress is interpreted as a scheduling directive that, when simulating execution of prototypes, allows experimentation to predict real-time behavior. The **rate** construct can also be viewed as abstractly specifying the distribution of computational work, or dually can be regarded as abstractly specifying the distribution of computational resources needed to achieve that work. By prescribing expected work, rates abstractly specify the allocation of processor-time to processes needed to achieve that work. In a single processor setting, rates can thus dictate the relative percentage of time to receive in multiprocessing. In a parallel setting, rates abstractly specify the distribution of tasks to processors (and multiprocessing ratios) needed to achieve that work, effected for example by load balancing. The rate construct thus provides a flexible abstraction for controlling computational effort.

4. Applications of the rate construct

Several examples are now presented to illustrate the utility of the rate construct for a broad range of problems, demonstrating its expressive power for prototyping fundamental algorithms.

4.1. Minimizing parallel work

The first illustration concerns the problem of minimizing work in such problems as searching for a goal along several paths. In many robotic and other classes of search problems, the cost of search is proportional to the distance traveled in the search. The problem of searching in an unknown environment with this cost function is abstractly captured by the w-lane cow path problem [RKT92]. The name comes from the following scenario: a cow, Bessie, is standing at a crossroads with w paths leading off into unknown territory. On one of the paths there is a grazing field (the goal) at distance n from the intersection, and all other paths go on forever. Bessie's eyesight is bad so she can't know if she's on an endless path. With no prior knowledge of which path the field is on, the problem is to determine how she can find the field while traveling the least distance possible. This is an important problem in such areas as robotics, for example a robot in an unknown environment searching to get around an obstacle. Another important application is when there are multiple algorithmic solutions to a problem, and we are not sure which solution needs to be utilized or is optimal. In general the rate construct can be used to succinctly express complex notions of progress in more sophisticated randomized algorithms. For example, algorithms for the cow-path problem can use randomized techniques to achieve optimal work [RKT92].

A sequential solution to the cow-path problem might use depth-first iterative deepening, which iteratively explores each path an increasing finite depth before returning to the pasture. A parallel solution, however, would explore each path at the same time, using one process per path. The search stops when one process finds the goal, and its solution is reported as the shortest path. For this technique to minimize parallel time and correctly report the shortest path, an assumption is made of equal progress. In the case when more is known about the environment, for example that different paths have different likelihoods of success (i.e., being shortest), the progress might be weighted in order to increase the probability of finding the shortest path in minimal time.

A concrete instance of this problem is the earliest termination of two different sorting strategies to be run in parallel, with the objective of capturing the best time sorting behavior (i.e., the minimal duration). For example, some weighted execution r_Q/r_M of Quicksort and Mergesort might be desired in order to balance the large variance of the former with the higher complexity constant (but 0 variance) of the latter. Such a solution can be expressed in *Proteus* as:

```
( signal done(Quicksort(x))  ||
  signal done(Mergesort(x)) )
rate[r_Q, r_M] on done(y) (solution := y)
```

This program uses the parameterized exception handling constructs of *Proteus* in order to raise a signal when either process finishes and thus achieve early termination. It should be noted that, for the case of a single processor, the rate construct is tantamount to weighted-time interleaving.

4.2. Frequency of asynchronous iteration

The second example illustrates the way in which rates can succinctly specify different frequencies of iteration for asynchronous processes. Such a situation arises in problems such as many-body simulation in which independent portions of the computation perform integration using different time-steps. Many-body simulation is the key computational component in many challenging problems such as fluid mechanics and molecular dynamics simulation; the potential benefits of the latter include computer aided drug design and protein structure determination. In N-body simulation the goal is to simulate for a collection of N particles distributed in space the motion over time due to gravitational or electrostatic interaction between the particles. The naive solution requires N^2 comparisons to compute forces arising from pairwise interaction. More sophisticated algorithms rely on approximation of the lesser effects of far-away clusters of particles (perhaps modeling them by a few large particles), and on multigrid techniques which exploit this approximation by hierarchically decomposing the particle space into near and far-away points in order to isolate these "far-field" interactions [Gre87].

A further optimization is to, for a given particle, compute interactions with far-away points less frequently since their effects fall off rapidly with distance. Such a technique is used for example in the Generalized Verlet Algorithm described in [GHWS91], where particles are separated into *distance classes* and interactions with far-away particles are computed less frequently. The rate construct can be used to control this iteration frequency for clusters which may be running on asynchronous processes. One may also use a specification of rates of progress to effect higher frequencies of iteration for well-separated clusters which have high densities and thus must have small motion integration steps to accommodate higher acceleration.

4.3. Load balancing and data locality

Another potential use of the rate constructs is to abstract to some degree the requirements for load-balancing in adaptive solutions to irregular problems. For example, one variant of N-body simulation is an algorithmic refinement in which recursive spatial decomposition is adaptive, in the sense that cells are subdivided into subcells for use in far-field approximation only if the number of bodies in a cell is above a threshold. By specifying a rate for cell processes at each level of decomposition proportional to the number of bodies (or estimated work required for those bodies), one can direct processors to those cells with the most computational demands. There are implementations of adaptive partitioning and scheduling mechanisms which make explicit at a lower-level the recognition of

this work distribution in adaptive N-body simulation [SHT+92]. By specifying at a high-level this anticipated work distribution, the rate construct provides a flexible abstraction for controlling computational effort.

Rates thus provide a convenient abstraction for specifying work distribution which does not require the user to explicitly specify the process to processor mapping but instead lets the system determine such details. However, one important consideration is how to ensure that data locality is preserved under any load balancing performed to effect progress constraints. It is a challenging problem to integrate abstractions for specifying data locality with abstractions such as rates which may effect load balancing. The private variables in *Proteus* [MNPR92] provide a weak measure of specifying a hierarchical form of data locality. We are investigating other techniques to specify a more general locality topology.

5. Real-time constraints

In addition to progress constraints, *Proteus* provides constructs for specifying and analyzing real-time behavior that include temporal constraints that the environment imposes on a task, such as limits on response time, and temporal constraints that the task imposes on the environment, such as periodic behavior. The temporal constraints are to be interpreted as prescriptions which must be enforced by the program through scheduling directives; provision is made for handling exceptions of temporal constraints, for example timeouts. Among the *Proteus* primitives are:

duration t (stmt)	[event duration]
wait t	[delay]
wait t → (stmt)	[guarded timeouts]
every t (stmt)	[periodicity]
within t (stmt)	
on timeout (stmt)	[response time]
S1 **on** E F1()	[exception handlers]

These primitives encompass conventional real-time constructs such as those found in [GL91].

Our measurement of time relies fundamentally on viewing clocks as modules. Time parameters represent ticks on some named virtual clock, which is updated according to an underlying model of time and computation related to that found in [LVD+91]. A "clock" is just a module with an active process responsible for updating the clock and distributing its values to a set of processes which it is said to "time". A clock times those processes which are visible from its module as determined by standard scoping rules. Thus, several clocks may time one process, and several processes may be timed by one clock. The **duration** constraints provide a mechanism for associating execution times with code sections which can be used in controlling relative progress. For example, fixed-rate can be regarded as automatically generating a local "rate" clock for each process which tracks progress.

6. Resource configuration

Real-time behavior will depend not only on synchronization and time constraints, but also on resource requests such as demands for computation, on the resource configuration which maps requesting agents such as processes to resources such as processors, and on scheduling which attempts to resolve resource contention and satisfy requests [GL91]. All these factors – resources, requestors, constraints, and scheduling – interplay to determine real-time behavior, and both rate constraints and resource configuration can play heavily in this determination. For example, rate constraints on the scheduling processes themselves can prove vital in guaranteeing good response time in distributed algorithms for resource allocation [RS82]. This algorithm uses probabilistic techniques to give real-time response for the allocation of resources in a distributed system, based on the ranges of relative rates of progress of non-equi-speed processes.

Moreover, it is often critical at the language level to make the resource configuration explicit in order to ensure precise real-time behavior. For example, concurrent processes which are configured to run on one processor using interleaving may not satisfy the same time constraints as processes running truly concurrently. In a similar manner, the underlying resource configuration determines the effect of directives such as rates and priorities. *Proteus* therefore provides a simple means of mandating resource configuration through **resource** and **assign** (@) constructs:

resource R_1, $R_2 = [i : i$ **in** $1..100]$;
$(P_1@R_1 \parallel P_2@R_1 \parallel P_3@R_2 \parallel P_4@R_2)$
rate [0.5, 0.5, 0.5, 0.5]

This example specifies R2 as a parallel resource, and will interleave P_1 and P_2 while running P_3 and P_4 truly concurrently. This notion of process-to-resource mapping can be generalized to accommodate more complex virtual topologies. To facilitate rapid prototyping, *Proteus* permits relaxed specification in that some or all of the specifications may be omitted, in which case any behavior consistent with the given constraints is admissible.

Proteus thus adopts a uniform resource-based view of time and progress which supports the expression of resource requirements and configuration. It is intended that *Proteus* support, in addition to these prescriptive mechanisms, declarative observational assertions about real-time behavior which can be used for verification at run-time or to check timing feasibility at compiler time. These declarative assertions appear in the specification side of the module, whereas directives appear in the implementation or body of the module, similar in style to [LVD+91].

7. Summary and future work

In this paper we have introduced the *rate* construct and examined its utility for a variety a problems such as minimizing parallel work in weighted parallel search and controlling the frequency of iteration in adaptive many-body simulation. Rates provide a valuable abstraction for exploring parallel algorithms at a level which ignores unnecessary detail. As such they form a promising addition to *Proteus*, which relies on

other high-level abstractions for interaction in shared-memory concurrency.

We are currently pursuing implementation of the rate construct on our sequential *Proteus* interpreter, to use in experiments with algorithmic variations of adaptive N-body simulation. We are also investigating extended models of resource-based declarations, including those which allow the specification of a hierarchical concept of nearness. We envision that, in conjunction with abstractions for data locality, the transformational approach which underlies architecture-independence in *Proteus* might be used to transform rate primitives to a somewhat more direct form of mapping and scheduling. The formal semantics and theoretical implications of restricting rates of computational progress is also under investigation, for example to determine its use in reasoning about real-time response and algorithms for resource-granting systems.

References

[Ble92] G. E. Blelloch, "NESL: A nested data-parallel language," Technical Report CMU-CS-92-103, Carnegie Mellon University, Jan. 1992.

[CCL91] M. C. Chen, Y. il Choo, and J. Li, "Crystal: Theory and pragmatics of generating efficient parallel code," in *Parallel Functional Languages and Compilers* (B. K. Szymanski, ed.), ch. 7, pp. 255–308, ACM Press, 1991.

[CZ89] R. Cole and O. Zajicek, "The APRAM: Incorporating asynchrony into the PRAM model," in *Proc. of the First ACM Symp. on Parallel Algorithms and Architectures*, pp. 169–178, ACM Press, 1989.

[Dyb87] R. K. Dybvig, *The SCHEME Programming Language*. Prentice-Hall, 1987.

[GHWS91] H. GrubMuller, H. Heller, A. Windemuth, and K. Schulten, "Generalized Verlet algorithm for efficient molecular dynamics simulations with long-range interactions," *Molecular Simulation*, vol. 6, pp. 121–142, 1991.

[GL91] R. Gerber and I. Lee, "Specification and analysis of resource-bound real-time systems," Technical Report MS-CIS-91-96, University of Pennsylvania, Nov. 1991.

[Gre87] L. F. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, 1987.

[KL91] K. Kenny and K. Lin, "Building flexible real-time systems using the Flex language," *IEEE Computer*, vol. 24, pp. 70–78, May 1991.

[Lam78] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, vol. 21, pp. 558–565, July 1978.

[LF81] N. A. Lynch and M. J. Fischer, "On describing the behavior and implementation of distributed systems," *Theoretical Computer Science*, vol. 13, pp. 17–43, 1981.

[LVD+91] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz, "Partial orderings of event sets and their application to prototyping concurrent timed systems," Technical Report, Computing Systems Laboratory, Stanford University, Dec. 1991.

[MNP+91] P. H. Mills, L. S. Nyland, J. F. Prins, J. H. Reif, and R. A. Wagner, "Prototyping parallel and distributed programs in *Proteus*," in *Proc. of the Third IEEE Symp. on Parallel and Distributed Processing*, (Dallas, Texas, Dec.1-5), pp. 10–19, IEEE, 1991.

[MNPR92] P. H. Mills, L. S. Nyland, J. F. Prins, and J. H. Reif, "Prototyping n-body simulation in *Proteus*," in *Proc. of the Sixth International Parallel Processing Symp.*, (Beverly Hills, Ca., Mar.23-26), pp. 476–482, IEEE, 1992.

[MO83] K. Marzullo and S. Owicki, "Maintaining the time in a distributed system," in *Proc. 2nd Symp. on Principles of Distributed Computing*, pp. 295–305, 1983.

[PP93] J. F. Prins and D. W. Palmer, "Transforming high-level data-parallel programs into vector operations," to appear in *Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (San Diego, Ca., May.20-22), May 1993.

[Ref88] Reasoning Systems, Inc., Palo Alto, California, *Refine 2.0 Language Summary*, Aug. 1988.

[RKT92] J. H. Reif, M. Kao, and S. Tate, "Searching in an unknown environment," in *Proc. of the Fourth Annual ACM-SIAM Symp. on Discrete Algorithms*, (San Diego, Ca.), ACM Press, 1992.

[RS82] J. H. Reif and P. Spirakis, "Real time resource allocation in distributed systems," in *Proc. of ACM Symp. on Principles of Distributed Computing*, (Ottawa, Canada), pp. 84–94, Aug. 1982.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets, An Introduction to SETL*. Springer-Verlag, 1986.

[SHT+92] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load balancing and data locality in hierarchical N-body methods," Technical Report CSL-TR-92-505, Stanford University, 1992.

[Ski90] D. Skillicorn, "Architecture-independent parallel computation," *IEEE Computer*, vol. 23, pp. 38–50, Dec. 1990.

[Ski91] D. Skillicorn, "Models for practical parallel computation," *International Journal of Parallel Programming*, vol. 20, no. 2, pp. 133–158, 1991.

[Smi90] D. R. Smith, "KIDS – a semi-automactic program development system," *IEEE Trans. on Software Engineering*, vol. 16, pp. 1024–1043, Sept. 1990.