The Computational Co-op: Gathering Clusters into a Metacomputer

Walfredo Cirne and Keith Marzullo

Computer Science and Engineering
University of California San Diego
http://www.cs.ucsd.edu/users/{walfredo,marzullo}

Abstract

We explore the creation of a metacomputer *by the aggregation of independent sites. Joining a metacomputer is voluntary, and hence it has to be an endeavor that mutually benefits all parties involved. We identify* proportional-share allocation *as a key component of such a mutual benefit. Proportional-share allocation is the basis for enforcing the agreement reached among the sites on how to use the metacomputer's resources. We introduce a* resource manager *that provides proportional-share allocation over a cluster of workstations, assuming applications to be master-slave. This manager is novel because it performs non-preemptive proportional scheduling of multiple processors. A prototype has been implemented and we report on preliminary results. Finally, we discuss how tickets (first-class entities that encapsulate allocation endowments) can be used in practice to enforce the metacomputer agreement, and also how they can ease the site selection to be performed by the application.*

1. Introduction

With commodity components dominating the computer industry, parallelism has become essential for high performance computation. The need for fast interconnection has made parallel super-computers the first feasible platform for parallel computation based on commodity components. In the past few years, however, advances in *local* networks have made it possible to use clusters of workstations as a more cost-effective platform for many parallel applications. Clusters can mimic parallel super-computers in many aspects, and are usually geographically compact and under a single administrative domain. Only very recently, advances in *wide-area* networks have made it possible to ensemble resources geographically dispersed across independent sites to execute some (primarily coarse grain) parallel applications. The execution of parallel application in such a wide-area setting has been called *metacomputing* [9].

The current efforts in metacomputing target many of the problems that arise in developing and running a parallel application over a wide-area infrastructure. However, they assume the user to have access to all sites her application is to use (*e.g.* Globus [8] and Legion [10]). From this somewhat user-centric viewpoint, metacomputing enables the user to effectively benefit from *all* resources the user has access to.

However, while a few influential users can manage to be granted with access to many different sites, most of them will only have access to a handful of sites. This is unfortunate because, besides being a great idea from the user point of view, metacomputing also makes sense from the site perspective. The basic motivation is *resource trading*: one offers local resources when they are available and obtains them remotely when the local demand exceeds the local capacity. From this standpoint, the role of metacomputing role is to "extend" sites. All users of a site would have potential access to many remote resources.

This paper focuses on how to turn the potential gain metacomputing can offer to sites into reality. Towards this goal, we present a resource manager for clusters of workstations that, assuming applications to be master-slave[1], provides proportional-share allocation[2] over the cluster as a whole. Proportional-share allocation supports organizing independent clusters of workstations into a metacomputer, called the Computational Co-op or simply the Co-op. Each cluster of workstation is thus called a Computational Co-op Member or simply a Co-op Member.

Section 2 discusses why proportional-share allocation is important for site-motivated metacomputing. It also presents the rationale for restricting sites to be clusters of workstations and applications to be master-slave. Section 3 describes the Co-op Member resource manager. Section 4 presents how its proportional-share allocation supports gluing multiple Co-op Members into one seamless Computational Co-op. Section 5 addresses how proportional-share allocation eases the selection of clusters by the application. Section 6 compares our work with others. Section 7 presents final conclusions and describes our plans for future work.

---

[1] Master-slave applications are parallel applications for which one process (the *master*) controls the computation.

[2] Proportional-share allocation enables the resource administrator to specify which fraction of the resource each user is to receive [18].

2. Requirements and Assumptions

Since sites are autonomous, each site has to perceive joining the Co-op as advantageous. Therefore, in order to promote the association of sites into a Co-op, *we need to provide a way to control the amount of resources each site contributes to and receives from the Co-op*.

Site administrators will want to enforce their resource usage policy towards the Co-op by determining how much of their resources are allocated to whom. This ability is necessary to enable site administrators to negotiate how much resource each site contributes to the Co-op, as well as how much of the aggregated Co-op resources will be available to each site. Such ability is called proportional-share resource allocation.

This argument supporting proportional-share allocation applies to any site-motivated metacomputing effort. In addition to this, we add two assumptions based on the coarse grain nature of metacomputing applications. We assume that applications are master-slave (a. k. a. *master-worker* or *manager-worker*). The master assigns pieces of computations to the others processes (the *slaves*), controls their progress, collects their outputs, and consolidates the outputs into the final result. Master-slave applications are regarded as being appropriate for the metacomputing environment due to their relatively low networking requirements [15]. Moreover, a large number of real-life applications are master-slave [3, 13, 14, 15]. In fact, our work started by examining the needs of the Nile project, whose goal is to provide wide-area support for the execution of computationally intensive high-energy physics applications [12], which are master-slave in structure.

We assume a Co-op Member to be a cluster of workstations because if an application is coarse grain enough to run in a metacomputer, then it should also run in a cluster with no problem. Of course, we could have chosen the Co-op Member to be a workstation by itself. However, this would largely increase the number of Co-op Members, making resource selection much more difficult.

3. The Computational Co-op Member

Tickets[3] provide a convenient way to express how much resource each application is to receive. Each application on the system owns a certain number of tickets that determines the resources it is to obtain: the fraction of the allocated resources assigned to an application is to equal the fraction of tickets it owns. Formally, letting t_i be the tickets a_i owns and r_i be the amount of resource a_i has received, we say the resource allocation is fair when $\forall a_i, a_j : \frac{r_i}{r_j} = \frac{t_i}{t_j}$, or, equivalently, when $\forall a_i, a_j : \frac{r_i}{t_i} = \frac{r_j}{t_j}$. Summing the first equation over j gives a system-wide condition: $\forall a_i : \frac{r_i}{\sum_x r_x} = \frac{t_i}{\sum_x t_x}$.

Besides being effective in determining how much resource each application is to receive, tickets turn allocation endowment into a first-class entity. For example, tickets can be granted to group leaders, each of which can distribute the tickets among users, each of which in turn determines how many tickets each application receives. This first-class feature is fundamental for Co-op creation: a Co-op is negotiated at the Member level and thus the negotiated allocations need to be redistributed to the users.

Stride scheduler [19] is an elegant and efficient proportional-share resource manager for uniprocessors. It allocates the next quantum of CPU time to the application a_i with the least r_i / t_i [4]. Since no more than one quantum is allocated at once (*i.e.* the stride scheduler is preemptive), this strategy guarantees that the relative allocations for any pair of applications never exceeds one quantum (*i.e.* $\forall a_i, a_j : \left| \frac{r_i}{t_i} - \frac{r_j}{t_j} \right| < 1$).

Therefore, it is natural to try to extend stride scheduling to the whole cluster by scheduling workstation by workstation in a circular fashion. In principle, there is no problem in centralizing the scheduling because (i) master-slave applications are coarse-grained, and (ii) the cluster is connected through fast and reliable local networking. However, stride scheduler is preemptive. The simple circular scheduling scheme could re-schedule a preempted application on another workstation, thereby requiring process migration. For applications that have any significant amount of state, process migration would very negatively impact application performance.

We address this problem in the simplest way: we don't preempt applications. Non-preemption matches well with master-slave applications because slaves are independent and consequently don't need to be running simultaneously. The master is the first process to run. After that, an application receives a workstation to run via a message to its master, which in turn starts the appropriate slave (using the Co-op Member remote execution support).

Unfortunately, not allowing preemption makes scheduling more difficult. The non-preemptive stride scheduler losses the guarantee that the relative allocation for any pair of application never exceeds one quantum. Adding the requirement to schedule multiple resources only makes things harder, because the scheduler cannot wait for running processes to finish (which is when the amount of resources consumed is known) before performing a new allocation. In fact, non-preemptive proportional-share scheduling of multiple resources is NP-hard, even when the execution times of the processes are known [4].

[3] *Tickets* were introduced by *lottery scheduling*, which is a seminal work on proportional-share scheduling [18].

[4] The ratio r_i / t_i is called pass in [19].

Hence, we provide an algorithm that asymptotically guarantees fairness and bounds how "unfair" the allocation can be at any point in time. Since how much resource the running processes are going to consume is unknown, we make allocations decisions based on the resources consumed *up to the moment* of the decision. When the first application a_1 starts, $r_1 = 0$. However, any subsequent application a_i should have $r_i = t_i \frac{r_j}{t_j}$, where a_j is the application already running with the least r_j / t_j. This makes a_i ready to be scheduled (since no other application a_j will have a smaller r_j / t_j), yet it does not give any advantage to a_i.

In order to bound the relative amount of the resource that one application can use over another, we assume that the cluster establishes a maximum p_{max} for the resources a single process can consume. A process is aborted if it reaches this limit. If a process is aborted, then the master is notified accordingly and can possibly start a number of (shorter-lived) slaves to replace the one that exceeded p_{max}. Putting this all together, Figure 1 summarizes the Co-op Member algorithm.

```
when application a_i starts at time t_i
    let a_j be the application with least r_j / t_j
    let r_i be t_i * r_j / t_j
when processor p becomes free
    for all applications a_x
        update r_x to reflect a_x's resource usage until now
    let a_i be the application with the least r_i / t_i
    run process p_{i,u} (of a_i) on p
when process p_{i,u} runs for p_max time units on p
    kill p_{i,u} (and thus free processor p)
```

Figure 1 – The Co-op Member algorithm

Since we avoid preemption, we cannot provide a fairness guarantee as tight as the stride scheduler. Instead, we guarantee precise fairness asymptotically and bound the skew between the resources actually allocated and the tickets proportions. We first show that p_{max} limits the allocation skew between any two applications.

Theorem 1: $\forall a_i, a_j : \frac{r_i}{t_i} \geq \frac{r_j - p_{max}}{t_j}$.

Proof: Assume the opposite, that is, $\exists a_i, a_j : \frac{r_i}{t_i} < \frac{r_j - p_{max}}{t_j}$. But, r_j cannot increase by more than p_{max} at once because this is the limit for the resources consumed by one process. Therefore, last time a_j received a processor p it was the case that $\frac{r_j - p_{max}}{t_j} \leq \frac{r_i}{t_i}$, otherwise a_i would have been chosen to receive p instead of a_j. **QED**

The pairwise bound on the allocation skew enables us to bound the amount of resources each application receives compared with the share of the resources it should receive.

Theorem 2: $\forall a_i : \frac{r_i}{\sum_x r_x} \geq \frac{t_i}{\sum_x t_x + n_a p_{max} \frac{t_i}{r_i}}$, where n_a stands for the number of applications in the system.

Proof: From Theorem 1, we have that $\frac{r_j}{r_i} \leq \frac{t_j}{t_i} + \frac{p_{max}}{r_i}$. Adding up for all a_j results in $\frac{\sum_x r_x}{r_i} \leq \frac{r_i \sum_x t_x + n_a p_{max} t_i}{t_i r_i}$, and consequently $\frac{r_i}{\sum_x r_x} \geq \frac{t_i}{\sum_x t_x + n_a p_{max} \frac{t_i}{r_i}}$. Since we have not fixed a_i, this formula holds for all a_i. **QED**

Theorem 3: $\forall a_i : \frac{r_i}{\sum_x r_x} \leq \frac{t_i}{\sum_x t_x + t_i \sum_x \varepsilon_x}$, where $\varepsilon_j = \frac{t_j^2 p_{max}}{t_i^2 r_i + t_i t_j p_{max}}$.

Proof: From Theorem 1, we have that $\frac{r_j}{r_i} \leq \frac{t_j}{t_i} + \frac{p_{max}}{r_i}$. Since this is valid for all a_i and a_j, we can exchange the indices and thus we have that $\frac{r_j}{r_i} \geq \frac{t_j r_j}{t_i r_j + p_{max} t_j} = \frac{t_j}{t_i} - \varepsilon_j$. Adding up for all a_j gives us $\frac{\sum_x r_x}{r_i} \geq \frac{\sum_x t_x}{t_i} - \sum_x \varepsilon_x$. This implies that $\frac{r_i}{\sum_x r_x} \leq \frac{t_i}{\sum_x t_x + t_i \sum_x \varepsilon_x}$. **QED**

Theorems 1-3 determine which factors affect the skew. In particular, p_{max} is under control of the site administrator and thus can be used to tighten the bounds if desired. Moreover, the bounds become progressively more tight as more resources are allocated, as shown in the following theorem.

Theorem 4: $\forall a_i : \lim_{r_i \to \infty} \frac{r_i}{\sum_x r_x} = \frac{t_i}{\sum_x t_x}$.

Proof: Considering that n_a, p_{max}, and t_i are constant, we have that $\lim_{r_i \to \infty} n_a p_{max} \frac{t_i}{r_i} = 0$. Together with Theorem 2, this implies that $\lim_{r_i \to \infty} \frac{r_i}{\sum_x r_x} \geq \frac{t_i}{\sum_x t_x}$. Similarly, all components of $t_i \sum_x \varepsilon_x$ are constant but r_i and thus

$\lim_{r_i \to \infty} t_i \sum_x \varepsilon_x = 0$. Together with Theorem 3, this implies that $\lim_{r_i \to \infty} \frac{r_i}{\sum_x r_x} \leq \frac{t_i}{\sum_x t_x}$. **QED**

The proofs above assume that applications always have more processes to run. In practice, however, applications finish and hence their level of parallelism impacts the bounds on fairness. For example, a one-process application with half of the tickets cannot get half of the resources of a three-machine cluster. In general, the problem is that when a_i is *terminating* (*i.e.* it does not need any more processors to finish, it only needs to finish the processes it is already running), no more processors can be allocated to it. Therefore, r_i / t_i may fall behind the r / t ratio of non-terminating applications. Unfortunately, there is nothing we can do about it. Allocating resources to terminating applications would leave such resources idle. That is, the bounds on resource allocation for terminating application could be maintained by *denying* resources to the other applications, a strategy that makes no sense in practice.

The Co-op Member algorithm has a number of practical advantages: (i) It requires no modifications to the underlying operating system. (ii) Processors do not need to be dedicated to the Co-op Member since the resource a process consumes is processor time. In fact, a Member's process can run with a lower priority if desired. (iii) Opportunistic computation (*i.e.* the utilization of cycles that would otherwise be wasted) can be supported by translating the lack of availability of a workstation into its failure. Of course, there are additional issues (such as checkpointing) that need to be addressed in such an environment [11]. (iv) Processors may be heterogeneous. One can support heterogeneous processors with a scale factor that makes the amount of resources consumed by a process processor-independent. (v) If a process is not able to fully use the resources of a processor, then another process can be started there. That is, the concept of a processor becoming free does not necessarily mean that it has no processes running. (vi) Because all communications and scheduling decisions are made only when a process finishes or when a processor is added to the cluster, the Co-op Member does not require clock synchronization and is a lightweight service. (vii) Adding and removing processors from the cluster are straightforward.

We have implemented a prototype of the Co-op Member resource manager in Java. We have run some preliminary experiments on a four-machine Sparc 5 cluster (with the resource manager running on another workstation). Our first experiment consists of two identical applications with the same number of processes (40) and equal number of tickets. Processes are a synthetic benchmark that reproduces the behavior of a high-energy physics application. They run for 15 seconds on a Sparc 5. This benchmark was developed by the Nile project to be used to evaluate design decisions. Figure 2 shows how resource gets allocated to the two applications. The data is gathered whenever a workstation is allocated.

It is interesting to notice that the Co-op Member often allocates the whole cluster to one application. It is this behavior that produces the staircase effect of Figure 2. This happens because the applications are identical, start simultaneously, and their processes always take roughly the same amount of time. All together, these factors imply that all workstations are allocated and subsequently freed at roughly the same time. Therefore, the application that is behind (*i.e.* has smaller r_i / t_i) receives the whole cluster (recall that allocation is based on the resources consumed up to the decision moment). In practice, however, processes run for different amounts of time, applications are distinct and start independently. In order to investigate a more realistic setting, we make the processes' execution time randomly vary from 15 seconds to one minute. As expected, the allocation becomes smother, as shown in Figure 3.

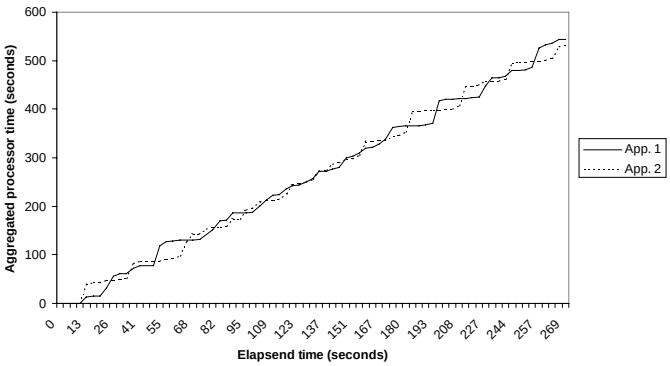

Figure 2 – Resource allocation for two identical applications

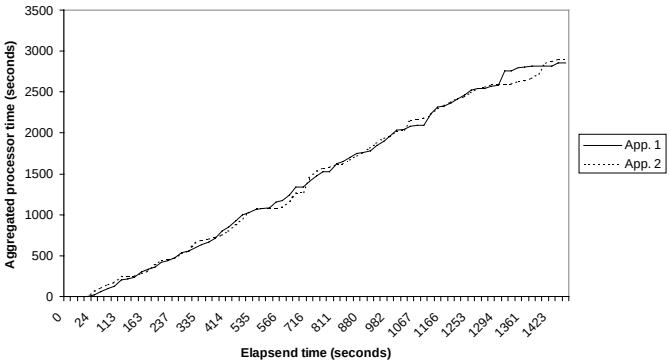

Figure 3 - Resource allocation for two identical applications whose processes' execution time varies

Of course, the allocation obeys non-even ticket proportions. Figure 4 shows the resources allocated to two appli-

cations whose ticket ratio is 4:1. The applications are otherwise identical to those of Figures 2.

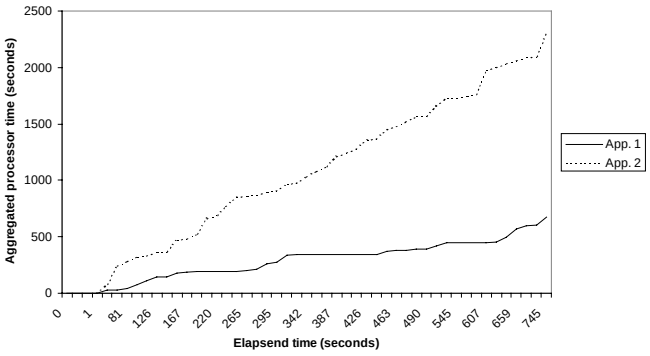

Figure 4 – Resource allocation for applications with 4:1 ticket ratio

4. The Computational Co-op

The Co-op Member's ability to perform proportional-share allocation relies upon the *relative* amount of tickets each user owns. Changing the number of tickets a user owns (or creating a new user with an initial ticket allocation) therefore affects the share of resources. This represents a problem because it affects the fraction of resources a Member contributes to the Co-op, and such a fraction is the basis of the Co-op agreement. Therefore, it is necessary to group ticket holders in a way that changing the relative ticket distribution has effects only within the group. This need is fulfilled by *ticket currencies*, a mechanism for modular ticket management proposed by the lottery scheduler [18]. The idea is that tickets are issued under a currency. Each Member has a base currency. One uses tickets from established currencies to back up a new currency; these tickets *fund* the new currency. The scheduler works only with base tickets; all non-base tickets are proportionally converted to their funding tickets until the equivalent in base tickets is obtained.

Currencies make it possible for each Co-op Member to create a local currency and a Co-op currency. The local currency provides tickets to the users and thus defines the local resource-sharing policy. The Co-op currency provides tickets to be exchanged with others Co-op Members. In summary, currencies enable sites to negotiate a fraction of their resources while keeping total flexibility in the allocation of tickets to local users.

A Member receives tickets from other Members during the creation of the Co-op. However, the Member is not the final user of such resources; its users are. Therefore, the Member has to transfer the external tickets it has to its users. This *ticket transfer* ability is what actually turns tickets into first class entities that encapsulate the right to use a resource.

With ticket transfers, tickets become a sort of e-cash. In fact, all schemes to ensure the correct use of e-cash can be directly applied. In our prototype, however, we implement a simple scheme for ticket transfer by storing the tickets in the Co-op Member resource manager. The ticket holder contacts the Co-op Member and informs how many of her tickets should be transferred to whom.

Notice that the way a member redistributes its external tickets is an administrative issue. The Co-op mechanisms do not determine it. One possible way would be to statically transfer the external tickets to the users. Another way would be to dynamically divide the external tickets among the active users, maybe following the policy for sharing of local resources (which is specified by the distribution of local tickets). This second way is probably better because it maximizes the usage of external resources.

5. Application Scheduling on the Computational Co-op

The Co-op Member resource manager makes all scheduling decisions (*i.e.* who runs where and when) within the cluster. However, the application (or some scheduling agent acting on its behalf) decides which clusters to use. There is no global scheduling entity because, due to the non-uniform distribution of resources across the sites, good scheduling decisions tend to be application-specific in the wide-area setting (*e.g.* [1]). For example, the data a particular application is interested on is likely not to be available on all sites, and its transfer over wide-area channels may take a prohibitively long time. Furthermore, there might be other application-specific constrains such as avoiding sites deemed as having weak security. In fact, recent research in metacomputing scheduling suggests that the application should be in charge of scheduling decisions such as process placement [2].

On the other hand, the gains of application-specific selection of the workstations within a site are much smaller because (i) they have fast networking connecting them (which reduces the importance of data transfer issues), and (ii) they are under the same administrative domain, which homogenizes administrative issues (such as security). Additionally, having applications selecting resources on a workstation per workstation basis would multiply the number of resources by at least one order of magnitude and therefore increase considerably the complexity of resource selection. Consequently, we believe that dividing the scheduling responsibility between application and system support at the site level is a good trade-off in our setting.

Information about the system performance is crucial to enable effective application-level scheduling [2]. In most systems it is rather difficult to obtain accurate information about the expected system performance [20]. In the Computational Co-op, conversely, tickets provide a straightforward way to gauge the performance a particular Co-op Member will deliver to an application. On one hand, tick-

ets are a lower-bound guarantee (when one considers the total number of tickets and the maximum skew). On another, they provide a very good estimate on how much resource the application is going to receive (when one considers the number of tickets owned by active applications). Of course, an application can be notified when there are changes on the total number of active tickets in some site (an event that impacts the amount of resource such site gives to the application) and adapt accordingly. The master-slave structure helps in such adaptation. In fact, an application can adapt at very low cost by simply changing the way it was going to start slaves in the various sites.

6. Related Work

Proportional-share scheduling has been receiving considerable attention in the past few years. However, we do not know of any work that uses the ability to precisely control resource allocation as the basis for building a metacomputer. Moreover, most of the work in proportional-share scheduling primarily targets uniprocessors. An exception is the work of Stoica and al. [16, 17]. They have proposed an economic-inspired proportional-share scheduler for parallel computers. However, they assume a homogeneous and dedicated environment, which is adequate for modeling parallel super-computers but not for clusters of workstations.

There have also been a few studies that schedule parallel applications over a cluster on which each workstation is managed by a proportional-share resource manager [6, 21]. Proportional-share can be very useful in this scenario by (i) using ticket information to steer process placement [6], (ii) fairly allocating resources among very distinct competing applications (*e.g.* sequential interactive and parallel batch) [6], and (iii) enhancing the predictability of the workstation scheduling, which enables a low-overhead gang scheduling scheme [21]. Note that points (i) and (iii) are related and also explored in our work. Indeed, the Computational Co-op makes the allocation of cluster's resources more predictable, thereby simplifying the selection (and possible reselection) of the best clusters by the application. In this regard, these approaches differ from the Computational Co-op in that our proportional-share allocation covers the whole cluster (instead of a workstation), enhancing thus the system's scalability.

Metacomputing appeared as a research area only a few years ago. Nevertheless, it has drawn a lot of attention and it is the focus of numerous ongoing research projects (*e.g.* AppLeS [2], Flock of Condors [7], Globus [8], and Legion [10]). The Computational Co-op is somewhat similar to the Flock of Condors in the sense that autonomous sites form a metacomputer envisioning mutual benefits. However, Condor focuses on harnessing idle cycles to support long-running high-throughput applications. Although the Computational Co-op could also be used in this scenario, its proportional-share allocation capabilities allows the precise definition of which fractions of the site resources are reserved for interactive use and for Co-op applications.

To the best of our knowledge, this is the first work that addresses how metacomputers are formed out of resources controlled by many independent entities, and how the users are endowed with the access to the metacomputer in such an environment. Elsewhere, the assumption is that the user has managed to acquire access to resources spread over different administrative domains, something that won't be the feasible to most users.

7. Conclusions and Future Work

This paper explores how *independent* sites can negotiate the sharing of their resources, and how the agreement reached in such a negotiation turns the agglomeration of sites into a metacomputer. In this scenario, joining the metacomputer is voluntary, and hence it has to be an endeavor that mutually benefits all parts involved. Due to these characteristics, we name our solution as the Computational Co-op, and refer to the sites as Computational Co-op Members.

Toward this goal, we identify proportional-share allocation as a key component for supporting the attraction of joining a Computational Co-op. Proportional-share allocation is the basis for enforcing the agreement reached among the Co-op Members on how to use the Co-op's resources. We then describe the Co-op Member resource manager, which provides proportional-share allocation over a cluster of workstations, assuming applications to be master-slave. Although based on the stride scheduler [19], the Co-op Member is novel because it addresses the non-preemptive scheduling of multiple processors. A prototype of the Co-op Member has been implemented and some preliminary results are shown.

Finally, we discuss how tickets are used in practice to enforce the Co-op agreement, and also how they ease the site selection decision to be performed by the application. In fact, in order to make sensible decisions about placement, applications need to know the performance each site can deliver, an information that can be precisely obtained from the relative number of tickets available for the application and the aggregated power of the cluster. Note also that basing the Computational Co-op in the exchange of tickets is a naturally distributed approach. There is no central entity to allocate the Co-op resources. This is especially desirable because Co-op Members are expected to be geographically dispersed, and thus any central entity would be a bottleneck in terms of performance and/or fault-tolerance. Moreover, a generic scheme for deciding which sites should be used would preclude the use of application-specific knowledge to improve scheduling.

In summary, the Computational Co-op Member manages the resources of a cluster in a way that (i) supports the assemble of independent clusters into a metacomputer

created by the negotiation of clusters' administrators, (ii) provides a simple way to assign access rights to users spread across the sites that form the metacomputer, and (iii) eases the development of application-centric schedulers by making the underlying infrastructure more predictable.

Of course, there are a number of aspects that deserve further attention. First, we need to evolve our prototype into a deployable version that addresses real-life concerns that we ignore at the moment (such as security). We plan to use Globus services to ease such a task by extending the Globus Resource Allocation Manager [5] to support the Computational Co-op as the underlying resource manager. Second, we plan to further evaluate how much the predictability on resource allocation (provided by the Co-op Member) eases application scheduling (in our case, cluster selection), and also how much is gained by grouping the workstations of a cluster under a single proportional-share resource manager (in opposition to have an independent proportional-share resource manager per machine). Third, we intend to extend the Co-op scope to deal with resources other then processors (*e.g.* disk space).

Acknowledgements

Walfredo Cirne is currently on leave from the Departamento de Sistemas e Computação, Universidade Federal da Paraíba, Brazil to undertake his Ph.D. at UCSD. CAPES partially supports Walfredo Cirne at UCSD (grant BEX2428/95-4). This work is also funded in part by the NSF (grants ASC-9318151 and ASC-9701333).

Thanks also to Fran Berman, Marcio Faerman, Jaime Frey, Russell Impagliazzo, Shava Smallen, and Rich Wolski for their insightful comments on this paper.

References

[1] Alessandro Amoroso, Keith Marzullo, and Aleta Ricciardi. *Wide-Area Nile: A Case Study of a Wide-Area Data-Parallel Application*. 18th International Conference on Distributed Computing Systems (ICDCS'98), pp. 506-15, May 1998.

[2] Fran Berman and Rich Wolski. *The AppLeS Project: A Status Report*. 8th NEC Research Symposium, Berlin, Germany, May 1997.

[3] Nicholas Carriero, David Gelernter, David Kaminsky and Jeffery Westbrook. *Adaptive Parallelism with Piranha*. Technical Report 954, Yale University, Department of Computer Science, February 1993.

[4] Walfredo Cirne and Keith Marzullo. *The Computational Co-op: Gathering Clusters into a Metacomputer*. UCSD Technical Report CS99-611, January 1999.
www-cse.ucsd.edu/users/walfredo/resume.html#publications

[5] K. Czajkowski et al. *A Resource Management Architecture for Metacomputing Systems*. IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[6] Andrea Arpaci-Dusseau and David Culler. *Extending Proportional-Share Scheduling to a Network of Workstations*. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada, June 1997.

[7] D. Epema et al. *A Worldwide Flock of Condors: Load Sharing among Workstation Clusters.* Journal on Future Generations of Computer Systems, vol. 12, no. 1, Elsevier, p. 53-65, May 1996.

[8] Ian Foster and Carl Kesselman. *The Globus Project: A Status Report*. IPPS/SPDP'98 Heterogeneous Computing Workshop, p. 4-18, 1998.

[9] Ian Foster and Carl Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers. July 1998.

[10] Andrew Grimshaw et al. *The Legion Vision of a Worldwide Virtual Computer*. Communications of the ACM, vol. 40, no. 1, January 1997.

[11] Miron Livny and Rajesh Raman. *High-Throughput Resource Management*. In [9], p. 311-37, 1998.

[12] Keith Marzullo et al. *Nile: Wide-Area Computing for High Energy Physics*. Seventh ACM SIGOPS European Workshop, Connemara, Ireland, 2-4 September 1996.

[13] Hong H. Ong, Iyad Ajwa, and Paul S. Wang. *PvmJobs: A Generic Parallel Jobs Library for PVM*. Special Sessions on Parallel and Distributed Computing Technology on 1997 IEEE National Aerospace and Electronics Conference (NAECON'97), vol. 1, p. 165-72, 14-18 July 1997.

[14] Jim Pruyne and Miron Liny. *Interfacing Condor and PVM to Harness the Cycles of Workstation Clusters.* Journal on Future Generations of Computer Systems, vol. 12, no. 1, Elsevier, p. 67-85, May 1996.

[15] Neil Spring and Rich Wolski. *Application Level Scheduling of Gene Sequence Comparison on Metacomputers*. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 1998.

[16] Ion Stoica, Hussein Abdel-Wahab and Alex Pothen. *A Microeconomic Scheduler for Parallel Computers*. Load-balancing and Job Scheduling for Parallel Computers, IPPS'95 Workshop. Lecture Notes in Computer Science, vol. 949, Springer-Verlag, p. 200-218, April 1995.

[17] Ion Stoica and Alex Pothen. *A Robust and Flexible Microeconomic Scheduler for Parallel Computers*. Third International Conference on High Performance Computing, p. 406-12, Thiruvananthapuram, India, December 1996.

[18] Carl Waldspurger and William Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management*. First Symposium on Operating Systems Design and Implementation (OSDI'94), p. 1-11, November 1994.

[19] Carl Waldspurger and William Weihl. *Stride Scheduling: Deterministic Proportional-Share Resource Mangement*. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[20] Rich Wolski, Neil Spring, and Chris Peterson. *Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service*. Supercomputing'97, Nov. 1997.

[21] B. B. Zhou, R. P. Brent, D. Walsh, and K. Suzaki. *Job Scheduling Strategies for Networks of Workstations*. 4th Workshop on Job Scheduling Strategies for Parallel Processing, March 1998.