An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes

Chin LU and Sau-Ming LAU
Department of Computer Science and Engineering
The Chinese University of Hong Kong
E-mail: clu@cs.cuhk.edu.hk, s.lau@doc.ic.ac.uk

Abstract

We propose an adaptive load balancing algorithm for heterogeneous distributed systems. The algorithm intrinsically allows a batch of tasks to be relocated. The key of the algorithm is to transfer a suitable amount of processing demand from senders to receivers. This amount is determined dynamically during sender-receiver negotiations. Factors considered when this amount is determined include processing speeds of different nodes, the current load state of both sender and receiver, and the processing demands of tasks eligible for relocation. Composition of a task batch is modeled as a 0-1 Knapsack Problem. We also propose a load state measurement scheme which is designed particularly for heterogeneous systems.

1. Introduction

A *dynamic load balancing* (LB) algorithm in a distributed system tries to improve system throughput and task response time by using the current system load information to relocate application tasks among the nodes in the system [1]. Most existing dynamic LB algorithms are targeted for homogeneous systems [3, 9, 11]. They are inappropriate for heterogeneous systems where processing nodes may have seemingly different processing speeds.

We propose a dynamic LB algorithm *GR.batch*, designed specifically for heterogeneous systems subject to different classes of tasks with different processing requirements. *GR.batch* is based on three major components: (1) the *batch task assignment* approach which allows a number of tasks to be selected for each remote assignment [5, 6], (2) the *GR Protocol* which negotiates between a sender and a receiver on the amount of workload to be contained in a task batch, and (3) an adaptive symmetrically-initiated *location policy* which ensures responsiveness of algorithm initiation to system workload change.

A critical design issue in *GR.batch* is the *task batch composition* (TBC) scheme, which refers to the strategy employed in selecting the set of tasks from among some possible candidates to compose a task batch. The TBC scheme is modeled as a *0-1 Knapsack Problem* (01KP). This formulation allows each sender-receiver negotiation session to yield the maximum amount of workload transfer, thus reduces the overall network bandwidth consumptions and CPU overhead substantially. A greedy solution to the 01KP is used. Simulations show that *GR.batch* is adaptive towards different heterogeneous systems and can provide very good performance improvements.

The rest of the paper is organized as follows. Section 2 presents the heterogeneous system model. Section 3 presents the *GR.batch* algorithm and the 01KP TBC scheme. Evaluations of *GR.batch* based on simulation studies are presented in Section 4. Section 5 is the conclusion.

2. Heterogeneous System Model

We assume the distributed system has no shared memory. Message passing is the only communication mechanism between processing nodes. We also assume that the network is fault free, strongly connected, and messages are received in the order sent.

We assume that in our system, nodes differ only in terms of processing speed. Tasks can run in any node in the system. Thus, the heterogeneity of the system is characterized by the different processing speeds of nodes in the system. Suppose, there are m different node types. Each node type T_i, where i is in M and $M = \{1, 2, \ldots, m\}$, is represented by a 3-tuple (T_i, μ_i, ψ_i) with μ_i and ψ_i being the processing throughput and the number of nodes of type T_i, respectively. The processing node composition in the system can be represented as a set of 3-tuples: $\{(T_1, \mu_1, \psi_1), (T_2, \mu_2, \psi_2), \ldots, (T_m, \mu_m, \psi_m)\}$. The total number of nodes in the system is $\Psi = \sum_{i \in M} \psi_i$.

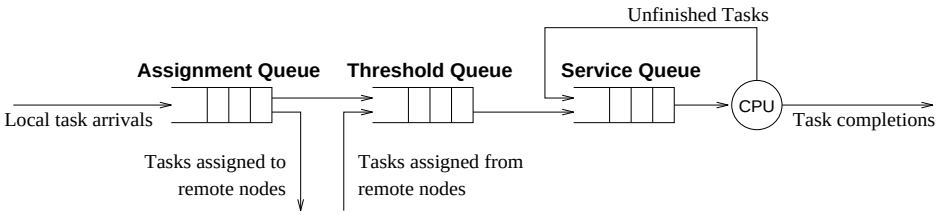


Figure 1. Model of a processing node

The node *id*s of type T_1 are $(P_1, P_2, \ldots, P_{\psi_1})$, and of type T_2 are $(P_{\psi_1+1}, P_{\psi_1+2}, \ldots, P_{\psi_1+\psi_2})$, and so on. We use the notation $T(P_j)$ to refer to the node type of a processing node having *id* P_j.

To model the relative processing speed of different nodes in a heterogeneous system, we define the *relative processing throughput* of node type T_i with respect to node type T_j as the ratio of the processing throughput of node type T_i to that T_j, and is denoted as μ_i^j,

$$\mu_i^j = \frac{\mu_i}{\mu_j}$$

For notational simplicity, the relative processing throughput between two *processing nodes* with *id*s P_x and P_y are also denoted as μ_x^y.

Based on relative processing throughputs, we define the *relative processing throughput matrix* $\Gamma = [\mu_i^j]$ as a Ψ by Ψ matrix, where μ_i^j is the relative processing throughput of node P_i with respect to node P_j. Γ is created during LB algorithm initiation so that μ_i^j can be obtained by simple table look up when needed.

Application tasks are categorized into n classes. Each task class C_i, where i is in J and $J = \{1, 2, \ldots, n\}$, is represented by three attributes. (1) *Service Demand*, denoted as w_i, is the processing requirement of a class C_i task. The *unit service demand* (one SD) represents the amount of computation that needs one second to complete with a particular node type T_R. The time needed to complete a task in other node types can be found easily by multiplying its service demand with the corresponding relative processing throughput. (2) *Code Length*, denoted as l_i, is the length in Kbytes of the task transfer message generated for a class C_i task if assigned remotely. (3) *Arrival Rate*, denoted as λ_i, is the Poisson local arrival rate of class C_i tasks to each individual node. The task class composition can then be represented as a set of 4-tuples: $\{(C_1, w_1, l_1, \lambda_1), (C_2, w_2, l_2, \lambda_2), \ldots, (C_n, w_n, l_n, \lambda_n)\}$. We use the notation $C(j)$ to refer to the task class of a task j.

It is essential to note that the task class specification scheme provides a systematic approach to the evaluations of different LB algorithms. The suitability and correctness of *GR.batch* are not in any way hindered by the task class specification scheme used.

The processing node model is shown in Figure 1. Each processing node consists of three queues: the *assignment queue*, the *threshold queue*, and the *service queue*. The assignment queue and the threshold queue both have infinite capacity, while the service queue has a fixed capacity. A node of type T_i has a service queue capacity denoted as Q_i.

When a local task arrives, it goes to the assignment queue to become a candidate for task assignment. The task may be assigned either locally by entering the local threshold queue, or remotely by entering the threshold queue on a remote node. Once a task enters a threshold queue, it cannot be reassigned to another node. That is, we do not allow *task reassignment*. A task in the threshold queue can be moved to the service queue only if the service queue has not reached its maximum capacity. Tasks in the service queue are processed by the CPU in round robin. A task runs on the CPU for at most a fixed time slice, T_{CPU}, each time. If a task cannot finish within its time slice, it goes back to the end of the service queue.

3. The *GR.batch* Algorithm

The core of *GR.batch* is the batch assignment approach, which allows a number of tasks to be selected for remote transfer during each sender-receiver negotiation session. Due to the differences in both processing throughputs and the amount of spare capacities, the amount of workload that a receiver can offload from its sender(s) varies tremendously. This makes the usual single task assignment approach inappropriate for a heterogeneous system. This also opens an opportunity for the batch assignment approach.

3.1. Location Policy

Location policy refers to the strategy used in finding an appropriate task transfer partner. The most commonly used approach is *polling*, which involves sending

a short query message to a selected target node to obtain a task transfer consent. Polling can be started either by a heavily loaded node in an attempt to locate a lightly loaded receiver, or vice versa. The polling in the former case is said to be *sender-initiated* and the latter *receiver-initiated* [10].

A study on dynamic LB algorithms done by Eager *et al* showed that neither pure *sender-* nor pure *receiver-initiated* location policy performs consistently over the whole range of system load [2]. In [9], Shivaratri and Krueger proposed an adaptive *symmetrically-initiated* algorithm in which sender-initiated pollings only occur at low system load, while receiver-initiated pollings are conducted whenever appropriate. This algorithm shows performance advantage over a wide range of system load. Thus *GR.batch* adopts the symmetrically initiated approach. *GR.batch* also uses polling limit (PL) to control the number of retry polls that a node can make [1, 4, 9, 10]. *GR.batch* is fully distributed in the sense that any node can initiate the algorithm based on the current system load.

3.2. GR Protocol

We designed the *Guarantee and Reservation Protocol* (GR Protocol) for the regulation of batch size. The GR Protocol is incorporated with exchanges of polling and task transfer messages. Each invocation of the GR Protocol is associated with two variables gur and res, where gur stores the amount of task service demands the sender is willing to transfer; and res stores the amount of task service demands the receiver is willing to accept. Details of how to obtain and update these two variables will be discussed in Section 3.4.

The basic idea of the GR Protocol is that the polling message sent from a sender to its target receiver node is piggybacked with gur. After a polling message is sent out, the workload of the sender will be reduced artificially by the amount equals to that of gur. This way, the sender pretends the workload in the amount of gur is already being catered for by the intended receiver and the sender itself becomes less busy virtually, as far as load state measurement is concerned. Based on gur from the polling sender, the receiver determines the value of res, which is piggybacked in the reply message to the sender. The sender then determines the final batch size and composes the task batch accordingly. res at the receiver node serves as a "quota" to reserve the receiver's processing capacity on behalf of the sender by elevating its load by the amount of res. In other words, the receiver node acts as if the sender's task batch is already running on it. This effectively avoids other senders from sending surplus workload to the receiver, which otherwise may become flooded. Thus, *processor thrashing* is avoided [5]. Once the actual task batch from the sender is received, the quota on the receiver side is released, while the arrival of the task batch elevates its actual workload.

As symmetrically-initiated location policy is used, a receiver can also start a polling session. The polling messages sent from the receiver to a target sender also carries the reservation value res. However, the derivation of its res value on the receiver side is entirely internal to the receiver and does not depend on any sender's guarantee value gur, which is simply not available. The rest of the protocol is identical to that of sender-initiated negotiations.

In order to disperse the workload in a heavily loaded node as quickly as possible and to fully utilize the capacities in a lightly loaded node, we allow multiple concurrent sessions of the protocol on the same node negotiating with different potential receivers/senders. Therefore, we need to define two variables that maintain the accumulated values of gur and res. We define the *accumulated reservation value* of a node P_i, denoted as R_i, as the total service demands that node P_i has currently reserved on behalf of all its sender partners divided by $\mu_{T(P_i)}$. It is the sum of all P_i's outstanding res values divided by $\mu_{T(P_i)}$. We also define *accumulated guarantee value* of a node P_i, denoted as G_i, as the total service demands that node P_i has currently guaranteed to transfer to all its receiver partners divided by $\mu_{T(P_i)}$. It is the sum of P_i's outstanding gur values divided by $\mu_{T(P_i)}$.

3.3. Workload Measurement

When there is only one task class, the workload state of a node can usually be measured by the number of tasks residing in the node. However, with multiple task classes, some tasks have larger service demands than others. Furthermore, the same service demand means different consumption of CPU time in different node types. These make the "number of tasks" index commonly used for homogeneous systems inappropriate. We define the *workload* of a node P_i, denoted by W_i, as the sum of the remaining service demands of the tasks residing in P_i, divided by P_i's processing throughput. That is,

$$W_i = \frac{1}{\mu_{T(P_i)}} \sum_{j \in K_i} w_{C(j)}$$

where K_i is the set of tasks residing in P_i. Division by $\mu_{T(P_i)}$ converts SD units to CPU time. Intuitively, W_i represents the time needed for P_i to complete all its residence tasks. When a task arrives, either locally or remotely, the service demands of the task divided by $\mu_{T(P_i)}$ is added to W_i. Conversely, when the task is

assigned remotely, or when the task has completed its execution, the same amount is deducted from W_i. We further define the workload of the assignment queue of node P_i, denoted as W_i^{AQ}, as the sum of the service demands of the tasks residing in the assignment queue of P_i, divided by P_i's processing throughput.

Based on the workload of a node P_i, we can define its *virtual load*, denoted by VW_i, as the workload of P_i plus its accumulated reservation value and minus its accumulated guarantee value. That is,

$$VW_i = W_i + R_i - G_i$$

Virtual load represents the committed workload of P_i.

According to [1, 8], a 3-level scheme is sufficient to represent the load state of a node. In *GR.batch*, nodes are assigned with three different load states according to their virtual load. The scheme is shown in Table 1. T_{up} and T_{low} are algorithm design parameters and are called *upper* and *lower threshold* respectively. By including the factor $1/\mu_{T(P_x)}$, W_x, R_x, G_x and thus VW_x are expressed in terms of CPU time rather than service demands. In this way, T_{up} and T_{low} provide a unified way for measuring workload state, regardless the processing throughput of individual node type.

Table 1. 3-level load measurement scheme

Load State	*Meaning*	*Criteria*
L-load	Lightly-loaded	$VW_i \leq T_{low}$
N-load	Normally-loaded	$T_{low} < VW_i \leq T_{up}$
H-load	Heavily-loaded	$VW_i > Tup$

3.4. Determining Batch Size

The strategy for deriving the guarantee value *gur* on the sender side has a profound effect on the performance of *GR.batch*. An exceedingly large *gur* prevents a sender from sending its surplus tasks to other potential receivers except the one currently in negotiation. Subsequently the amount of workload distribution will be limited. In contrast, a small *gur* limits the amount of workload transfer in a single batch, thus depriving the advantage of batch assignment.

The scheme for deriving *gur* is the following:

$$gur^{(s)} = \min \begin{cases} (T_{up} - T_{low}) \cdot \mu_r^s \\ W_s^{AQ} \end{cases} \quad (1)$$

The superscript $^{(s)}$ refers to sender node P_s. For example, $gur^{(s)} = 2.5$ means that the workload needs 2.5 time unit to complete if executed in P_s. (If $\mu_{T(P_S)} = 10$, $gur^{(s)} = 2.5$ then equals 25 SD.) To justify the validity of (1), we introduce the first regulation rule.

Rule 1 *The arrival of a task batch having a total service demand $\alpha^{(r)}$ should not boost up the receiver P_r to the H-load state. That is,*

$$\neg\ (VW_r + \alpha^{(r)} \Rightarrow \textit{H-load})$$

By the 3-level load measurement scheme defined in Table 1, Rule 1 gives $(VW_r + \alpha^{(r)}) \leq T_{up}$. Thus,

$$\alpha^{(r)} \leq (T_{up} - VW_r) \quad (2)$$

Again from Table 1, virtual load of a receiver node must be between 0 and T_{low}. Thus, the two bounding values of VW_r are:

$$\begin{array}{ll} \text{Minimum:} & VW_r = 0 \\ \text{Maximum:} & VW_r = T_{low} \end{array}$$

We then have the two bounding conditions of (2).

$$\alpha^{(r)} \leq \begin{cases} T_{up} & \text{if } VW_r = 0 \\ T_{up} - T_{low} & \text{if } VW_r = T_{low} \end{cases} \quad (3)$$

Since the second case in (3) gives a tighter bound, to take the less optimistic approach, sender P_s can expect that the amount of workload that receiver P_r will agree to receive is *at most* $(T_{up} - T_{low})$. This amounts to $(T_{up} - T_{low}) \cdot \mu_r^s$ when expressed in P_s's CPU time. Therefore, $gur^{(s)}$ should not be greater than this value. This explains the first criterion in (1). The second criterion in (1) is obvious as sender P_s cannot guarantee more than it has in its assignment queue.

When $gur^{(s)}$ arrives, receiver P_r has to derive the corresponding reservation value *res*. According to Rule 1 and neglecting possible local task arrivals before the actual task batch is received, P_r affirms inequality (2). Taking the largest possible value, we have

$$\alpha^{(r)} = T_{up} - VW_r \quad (4)$$

Obviously, P_r should not reserve its processing capacity by an amount greater than its sender has guaranteed. We therefore have

$$res^{(r)} = \min \begin{cases} \alpha^{(r)} = T_{up} - VW_r \\ gur^{(s)} \cdot \mu_s^r \end{cases} \quad (5)$$

Note however that from (1), the maximum value of $gur^{(s)}$ is $[(T_{up} - T_{low}) \cdot \mu_r^s]$, or $(T_{up} - T_{low})$ when measured in P_r's CPU time. On the other hand, since the maximum value of VW_r for receiver P_r is T_{low}, the minimum value of $\alpha^{(r)}$ is also $(T_{up} - T_{low})$. Thus, $gur^{(s)} \cdot \mu_s^r$ is always at least as constrainting as $\alpha^{(r)}$. Therefore, (5) can be simplified as $res^{(r)} = gur^{(s)} \cdot \mu_s^r$. For receiver-initiated pollings, by (4), we have $res^{(r)} = (T_{up} - VW_r)$ as $gur^{(s)}$ is simply unavailable. Thus,

$$res^{(r)} = \begin{cases} gur^{(s)} \cdot \mu_s^r & \text{if sender-initiated} \\ T_{up} - VW_r & \text{if receiver-initiated} \end{cases} \quad (6)$$

The next step in batch size regulation is for the sender P_s to determine the largest amount of task ser-

vice demands that are eligible to be transferred to the receiver P_r, under the constraint imposed by $res^{(r)}$. This amount is called the *maximally allowed batch size* and is denoted as θ. To derive θ, we introduce the second and the third regulation rules.

Rule 2 *The removal of a task batch having a total CPU time $\beta^{(s)}$ should not change the sender node P_s to the L-load state. That is,*

$$\neg\ (VW_s - \beta^{(s)} \Rightarrow \textit{L-load})$$

Again from Table 1, Rule 2 gives $(VW_s - \beta^{(s)} > T_{low})$. That is,

$$\beta^{(s)} < VW_s - T_{low} \tag{7}$$

Rule 3 *The relocation of a task batch having a total CPU time $\beta^{(s)}$ should not result in P_s having a lower virtual load than P_r. That is,*

$$\begin{aligned} VW_s - \beta^{(s)} &\geq VW_r^{'} + \beta^{(s)} \cdot \mu_s^r \\ \beta^{(s)} &\leq \frac{VW_s - VW_r^{'}}{(1+\mu_s^r)} \end{aligned} \tag{8}$$

where $VW_r^{'}$ is the virtual load of P_r as piggybacked on the ACK message from P_r to P_s in reply to P_s's polling.

By relaxing inequality (7) and taking the largest possible value satisfying (8), we have

$$\beta^{(s)} = \min \begin{cases} VW_s - T_{low} \\ \frac{VW_s - VW_r^{'}}{(1+\mu_s^r)} \end{cases} \tag{9}$$

Since sender P_s should not transfer more workload than the receiver has reserved, we therefore have

$$\theta = \mu_{T(P_s)} \cdot \min \begin{cases} \beta^{(s)} \\ res^{(r)} \cdot \mu_r^s \end{cases} \tag{10}$$

Note that θ is measured in SD units. From (6), the largest possible value of $res^{(r)}$ is T_{up} (receiver-initiated negotiation with $VW_r = 0$). It follows from (10) that the largest value of θ is $(T_{up} \cdot \mu_{T(P_r)})$. In other words, $(T_{up} \cdot \mu_{T(P_r)})$ is the upper bound of the size (in SD units) of a task batch that can be composed for P_r. The higher the processing throughput of a receiver, the larger the task batch possible for it.

After θ is determined, sender P_s has to select the tasks to be included into the final task batch according to the *task batch composition* (TBC) scheme. The *optimal* selection requires examining all tasks residing in the assignment queue. As this is an NP-complete problem as shown later, we must limit the number of candidates eligible for consideration by the TBC scheme. This limit is denoted as η. Only the first η tasks in the assignment queue are considered by the TBC scheme.

A TBC scheme can be formulated with different performance objectives. Such flexibility makes *GR.batch* applicable to a wide range of practical situations. In this paper, we choose the objective to be that it makes the maximal use of a given value of θ. In other words, it tries to transfer as much workload as allowed in each sender-receiver negotiation. This way, the relative CPU overhead and network bandwidth consumption spent in the negotiation protocol is reduced to a minimum. Since different task classes have different service demands, the TBC scheme is formulated as a 0-1 Knapsack Problem (01KP) [7]. As 01KP is NP-complete, it is essential to use a heuristic method to solve it so as to reduce the amount of runtime overhead. A greedy solution is thus proposed as follows.

The η candidate tasks are examined in sequence, starting from the head of the assignment queue. A task is inserted into the task batch if the remaining capacity is greater than or equal to the service demand of the task. The scanning process ceases until either θ is fulfilled completely or all candidate tasks have been examined. To guarantee a worst-case performance of 1/2, the task with the largest service demand is considered as a possible alternative solution if it yields better θ utilization.

4. Performance Evaluations

We assume that the CPU overhead for processing a polling message is non-negligible and is denoted as $CPU_{polling}$. Since a polling message is inherently short, we assume that only one message injection cost $F_{polling}$ is needed. The communication delay for transmitting a polling message is therefore defined as:

$$DELAY_{polling} = F_{polling} + D \tag{11}$$

where D is the network propagation delay.

We assume that both the sender and the receiver of a task batch are imposed with the same amount of CPU overhead, denoted by CPU_{task}, as defined below.

$$CPU_{task} = C_{LB} + C_{pack} * \sum_{i=1}^{b} l_{C(i)} \tag{12}$$

where C_{LB} and C_{pack} are constants representing the CPU time for running the LB algorithm, and for composing/decomposing each task message, respectively. The value b is the number of tasks in the task batch.

The communication delay for transmitting a task batch is also assumed to be non-negligible and is modeled as below.

$$DELAY_{task} = (F_{task} * \sum_{i=1}^{b} l_{C(i)}) + D \tag{13}$$

Table 2. Simulation parameter values — Calibrated to node type $T(P_1)$ where appropriate.

Parameter	*Value*	*Parameter*	*Value*	*Parameter*	*Value*	*Parameter*	*Value*
$Q_i, \ \forall\, i \in M$	30	PL	5	$CPU_{polling}$	0.005	D	0.010
T_{CPU}	0.1	$l_i, \ \forall\, i \in J$	5	$F_{polling}$	0.001	C_{pack}	0.001
T_{low}	5	η	50	F_{task}	0.001	C_{LB}	0.002
T_{up}	20						

where F_{task} is the time needed to inject a single task transfer messages into the communication channel.

4.1. Performance Metrics

The performance metrics we used are: mean task response time, percentage CPU overhead, net CPU utilization, batch size (the mean number of tasks contained in a task batch), and channel overhead. We also define receiver-initiated polling density, D_r, as follows.

$$D_r = \frac{1}{L \cdot \Psi} \sum_{1 \le i \le \Psi} R_i^{\rightarrow} \tag{14}$$

where $R_i^{\rightarrow}$ is the total number of receiver-initiated pollings created by node P_i during the measurement period L.

4.2. Performance Results

The values of simulation parameters used are given in Table 2. An adaptive algorithm, *SK.single*, which uses Shivaratri and Krueger's symmetrically-initiated location policy and allows single task assignment is selected for comparison purpose. Since both *GR.batch* and *SK.single* employ the same basic mechanism in their location policy, the comparisons allow easy visualization of the performance characteristics of *GR.batch*.

Homogeneous Task Class

We examine two different system types. Each system type contains ten processing nodes having different throughputs.

$$\text{Type-I:} \begin{cases} \mu &= \{1, 3, 5, 7, \cdots, 19\}, \\ \psi_i &= 1, \quad \forall\ 1 \le i \le 10 \\ \Psi &= 10 \end{cases}$$

$$\text{Type-II:} \begin{cases} \mu &= \{1, 2, 4, 8, \cdots, 512\}, \\ \psi_i &= 1, \quad \forall\ 1 \le i \le 10 \\ \Psi &= 10 \end{cases}$$

We examine a system in which only one task class exists. That is,

$$J_{homo} = \{(C_1, 1, 5, \text{ varying } \lambda)\}$$

Figure 2 shows the performance comparisons of *GR.batch* and *SK.single* in the two system types.

Observation 1: *GR.batch* **always gives a lower task response time.**

Figure 2(a) shows the mean task response time averaged over 10,000 tasks. For both Type-I and Type-II systems, *GR.batch* provides a far smaller response time than *SK.single* does over the whole range of task arrival rates studied. For Type-I system at $\lambda = 70$, a performance advantage of 25.7% is achieved. For Type-II system at the same task arrival rate, a performance advantage of 60% is observed.

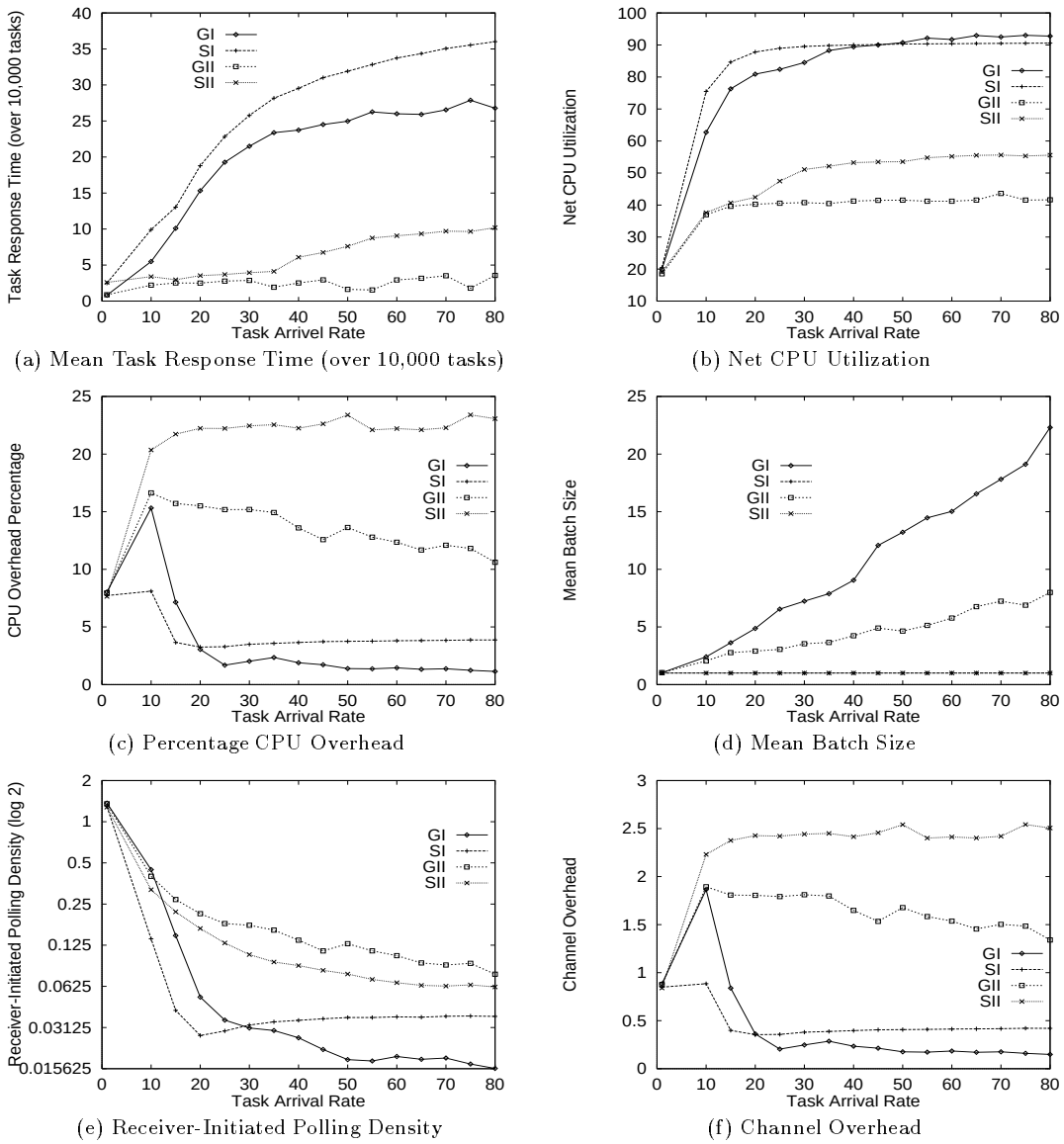
Observation 2: *GR.batch* **makes more efficient utilization of CPU capacity by favoring task transfer to fast processing nodes.**

Type-I: Figures 2(a) and 2(b) show that for system Type-I at task arrival rate below 35, *GR.batch* provides both lower net CPU utilization and task response time. This implies that when compared with *SK.single*, *GR.batch* transfers a larger portion of tasks to those fast processing nodes, running in which a task requires less processing time.

Type-II: Similar (but even more spectacular) performance characteristics as with Type-I is observed for Type-II. In fact, a reduction of 15% CPU utilization leads to a response time improvement of 57% at arrival rate 45 and 60% at arrival rate 70. These can be explained similarly as above and the larger performance improvement is due to the larger deviations of processing throughputs among the constituent nodes in Type-II.

Observation 3: *GR.batch* **imposes far less CPU overhead by efficient use and reduced number of negotiation sessions.**

Type-I: Figure 2(c) shows that below arrival rate 20, *GR.batch* has a larger CPU overhead than *SK.single*. For higher system loads, the reverse occurs. *GR.batch*'s lower CPU overhead ($\approx$ 50-70% reduction) at medium to high system loads can be attributed to the more efficient use of negotiation sessions. Suppose a task batch contains b tasks. To transfer the same amount of workload, *SK.single* needs b successful negotiation



(a) Mean Task Response Time (over 10,000 tasks)
(b) Net CPU Utilization
(c) Percentage CPU Overhead
(d) Mean Batch Size
(e) Receiver-Initiated Polling Density
(f) Channel Overhead

Figure 2. Performance comparisons — Homogeneous task class. G = $GR.batch$**, S =** $SK.single$**, I =** $Type-I$**, II =** $Type-II$

sessions. This means at least $(b-1)$ times more CPU overhead for polling, and $[(b-1) \cdot C_{LB}]$ more CPU overhead for task transfer. Since the mean task batch size grows with increasing system load, Figure 2(d), the difference in CPU overhead progressively enlarges.

Type-II: Figure 2(c) shows that $GR.batch$ exhibits a lower CPU overhead throughout the whole range of arrival rates studied. This can be explained similarly as above for Type-I system. On the other hand, it is found that the magnitude of CPU overhead in Type-II system is much higher than that in Type-I. Obviously, this is attributed to the larger amount of polling activities. When compared to Type-I system, receiver nodes in Type-II system can level out the surplus workload in senders more quickly because of their larger processing capacity. Receiver-initiated pollings in Type-II system more easily find themselves a failure and thus more polling sessions are created, Figure 2(e).

Observation 4: $GR.batch$ **imposes far less channel overhead by efficient use and reduced number of negotiation sessions.**

Channel overhead reflects the amount of polling activities and its similar trend as CPU overhead can be easily understood, Figure 2(f).

Multiple Task Classes

We define the third system type as follows.

$$\text{Type-III:} \begin{cases} \mu & = & \{1, 5, 10, 15, \cdots, 45\}, \\ \psi_i & = & 1, \quad \forall\ 1 \le i \le 10 \\ \Psi & = & 10 \end{cases}$$

We study three different cases where system Type-III is subjected with three different task arrival patterns shown below.

$$\begin{aligned} J_A &= \{(C_1, 1, 5, 0.1), (C_2, 10, 5, 0.5), (C_3, 30, 5, 0.5)\} \\ J_B &= \{(C_1, 1, 5, 0.9), (C_2, 10, 5, 0.5), (C_3, 30, 5, 0.5)\} \\ J_C &= \{(C_1, 1, 5, 0.1), (C_2, 50, 5, 0.8)\} \end{aligned}$$

The smallest task class C_1 in J_A has a small arrival rate while the other two classes have medium arrival rates. J_B differs from J_A only in the arrival rate of task class C_1. J_C represents a case in which large tasks are arriving very frequently, while there are some small tasks occasionally arrive into the system.

Table 3. Performance comparisons — Heterogeneous task classes.

		Response Time		CPU Overhead	
Task	Comp.	*GR.batch*	*SK.single*	*GR.batch*	*SK.single*
J_A	C_1	45.49	56.79	3.57	5.46
	C_2	26.66	37.04		
	C_3	45.77	52.89		
J_B	C_1	8.24	13.59	5.82	6.43
	C_2	16.80	22.39		
	C_3	36.84	37.65		
J_C	C_1	246.91	341.63	0.04	0.04
	C_2	245.97	292.83		

Table 3 shows the performance comparisons between the two algorithms in the three different cases. In all cases and for all task classes, task response time as exhibited by *GR.batch* is lower than that of *SK.single*. For example, task class C_1 has a performance advantage of 20% with task composition J_A, 38% with J_B, and 29% with J_C. It can be observed that a task class with smaller task service demand tends to enjoy a larger performance improvement. This shows that they are more frequently relocated to receiver nodes. Obviously, this can be attributed to their easiness to be "packed" into a task batch. Except in the case of J_C, the CPU overhead as produced by *GR.batch* is lower than that of *SK.single*.

5. Conclusion

We proposed the *GR.batch* algorithm designed specifically for heterogeneous distributed systems. The key of the algorithm is to transfer a suitable amount of processing workload from senders to receivers. The amount is determined dynamically by the GR protocol. Task batch composition is modeled as a 0-1 Knapsack problem and a greedy solution is employed. In addition, we also proposed virtual load as a workload measurement scheme.

The performance of the algorithm has been evaluated by simulations. The results showed (1) the adaptive behavior of the algorithm towards different heterogeneous systems; and (2) the performance improvement in terms of task response time and CPU and communication overhead.

References

[1] D. L. Eager and E. D. Lazowska. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, SE-12(5), May 1986.

[2] D. L. Eager and E. D. Lazowska. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[3] O. Kremien and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Trans. Distributed and Parallel Syst.*, 3(6):747–60, Nov. 1992.

[4] C. Lu and S.-M. Lau. A performance study on load balancing algorithms with process migration. In *Proceedings, IEEE TENCON 1994*, pages 357–64, Aug. 1994.

[5] C. Lu and S.-M. Lau. An adaptive algorithm for resolving processor thrashing in load distribution. *Concurrency: Practice and Experience*, 7(7):653–70, Oct. 1995.

[6] C. Lu and S.-M. Lau. An adaptive load balancing algorithm for systems with bursty task arrivals. In *Proceedings, Thirteenth IASTED International Conference for Applied Informatics*, Austria, Feb. 1995.

[7] S. Martello and P. Toth. *Knapsack Problems — Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[8] L. M. Ni, C. W. Xu, and T. B. Gendreau. Drafting algorithm - a dynamic process migration protocol for distributed systems. In *Proceedings, the 5th International Conference on Distributed Computing Systems*, pages 539–546. IEEE, 1985.

[9] N. G. Shivaratri and P. Krueger. Two adaptive location policies for global scheduling algorithms. In *Proceedings, The 10th International Conference on Distributed Computing Systems*, pages 502–9, May 1990.

[10] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Trans. Comput.*, C-34(3), Mar. 1985.

[11] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. Softw. Eng.*, SE-14(9):1327–41, Sept. 1988.