計算機アーキテクチャ　104－1
システムソフトウェアと
オペレーティング・システム　62－1
（１９９４．　１．　２７）

UNIXネットワークにおけるタスク分散システムの実現と性能評価

吉田 孝光　　高井 昌彰　　佐藤 義治

(北海道大学 工学部)

本論文では、分散システムにおけるコンピュータ間の負荷分散を行い、タスクの平均応答時間の短縮を図る比較的簡単な分散制御型の動的負荷分散方式を提案する。本方式は、負荷分散の対象となるタスクを移送が容易な未実行のものに限定することで実際の分散システムへの実装の可能性を高めるとともに、異機種を含む一般的な分散システムにも適用可能な負荷分散アルゴリズムを用いている。さらに本方式をNFSが実装されているUNIXネットワーク上にネットワークタスク分散システム(NTDS)として実現し、実験の結果から分散システムに負荷の偏りがある場合や適度な負荷が与えられた場合に負荷分散効果が顕著に現れることが確認された。

Implementation and Evaluation of the Network Task Distribution System on a UNIX network

Takahiko Yoshida, Yoshiaki Takai, and Yoshiharu Sato

Information and Graphics Sciences, Faculty of Engineering, Hokkaido University
North 13 West 8, Kita-Ku, Sapporo, 060, Japan

In this paper we propose a dynamic load balancing algorithm for distributed systems. The prime objective of our algorithm is to shorten the mean response (turnaround) time of batch tasks such as noninteractive simulation programs by balancing workloads among computers in a distributed system. We focus on a simple mechanism for allocating or migrating tasks before their execution, and implement it on a local area network composed of heterogeneous UNIX workstations as a Network Task Distribution System (NTDS). We show that the NTDS significantly improves the mean response time by experimental results.

1 はじめに

複数のコンピュータを LAN(Local Area Network) で接続することで、コンピュータ間で分散された二次記憶などの資源の共有が可能であり、このようなコンピュータシステムはその形態から一般に分散システムと呼ばれる。汎用コンピュータを頂点に据えた垂直型分散処理から、LAN による水平型分散処理へとコンピュータの利用形態が急速に変わってきている。

このような分散システムにおいて、接続されたコンピュータの能力をより有効に利用するには負荷分散の機能が要求されるが、近年このような負荷分散方式に関する研究が注目されており実際に数多くの手法が提案されてきた [1]-[5]。

従来の負荷分散方式に関する研究は、その多くがシミュレーションによる性能評価で実際の分散システムへの実装による評価を課題としているものが多いが、通信遅延の設定が非現実的と思われるものや、実装する際に重要な実行中のプロセス移送の問題などをあまり考慮していないものも少なくない。また、分散システムへの実装を行った研究もあるが [4]、同機種間のコンピュータによる評価であり異機種間の分散システムにおける評価までは行われていない。分散システムがもつ機能分散という利点を考慮すると、将来的には構成されるコンピュータがすべて同一機種であることは考えにくく、このような異機種間の分散システムにも適用できる負荷分散機構が望まれる。

そこで本研究では、負荷分散の対象となるタスクは未実行のものと限定し、分散システム上で負荷状況に応じてタスクを動的に分散させることによりタスクの平均応答時間の短縮を図る、異機種間の分散システムにも適用可能な動的負荷分散方式を提案する。さらに、本方式を実際にネットワークタスク分散システム (Network Task Distribution System, NTDS) として UNIX ワークステーションで構成される異機種間分散システム上へ実装し、実際にタスクを生成させる実験により本方式の性能評価を行う。

本研究は、分散の対象となるタスクを移送が容易な未実行のものと限定し、比較的簡単な負荷分散アルゴリズムを用い、UNIX などの既存のオペレーティングシステムにおいてアプリケーションレベルでタスクを分散させるシステムを構築することに主眼を置くものである。

2 諸定義

2.1 対象とする分散システム

本方式は、マルチタスク機能を有するオペレーティングシステムを搭載したコンピュータで構成される、分散ファイルシステムを利用した分散システムを前提としている。分散システムを構成するコンピュータは「ホストコンピュータ」と呼ばれることがあるが、本論文中では以降、簡単に「ホスト」と呼ぶ。ホストコンピュータとして使用するものは同じ機種で統一する必要はなく、プロセッサが異なってもオペレーティングシステムなどの機能に互換性があるものであればよい。そういう意味での異機種間の分散システムにも本方式は適用可能であり、実際に想定しているのは、UNIX ワークステーションをイーサネットでバス結合し、Sun Microsystems Inc. の NFS(Network File System) や AT&T の RFS(Remote File Sharing) などの分散ファイルシステムを利用した分散システムである。分散ファイルシステムを利用すれば、一般のユーザは通信ネットワークの存在をほとんど意識することなく、ネットワーク上の複数のホストに分散されたファイルシステムにアクセスすることができる。また、資源の共有という意味でも重要である。

2.2 想定するタスク

「タスク」と「プロセス」は同義語として扱われることが多いが、本論文中では「タスク」というとユーザから実行を依頼された仕事でまだ CPU による処理が開始されていないものを指し、CPU による処理が開始されたものを「プロセス」と呼ぶこととする。つまり、ユーザからの仕事はまず「タスク」という状態で CPU へ割当てられるのを待ち、実行が開始される (これを「ディスパッチされる」という) と「プロセス」になると考える。

本方式では、ネットワーク上の各ホストに、Network Task Distributor(NTD) という 1 つのプロセスを起動させておき、本方式を使用するユーザはこの NTD に対してタスクを投入するものとする。NTD に関する説明は後述する。本論文中では、「他のユーザによるタスク (あるいはプロセス)」という場合は、この NTD を通さず、直接オペレーティングシステム上のプロセスとして実行されたタスクとして定義する。以降、ただ単に「ユーザ」といった場合は本方式を使用するユーザを指すことにする。また、「ユーザ」は複数存在してもかまわないが、NTD は各ホスト上に 1 つだけ起動されるものとする。

ユーザは、タスクを投入する際、複数のタスクをまとめたバッチプログラムという形で NTD に投入する。NTD が扱うタスクには条件が 2 つある。第 1 は、互いに依存関係がなく実行順序などは問わないということ、第 2 は、そのタスクがする仕事はそれが実行されるホスト独自の情報を用いないということである。これらの条件を満たすタスクとしては、たとえば、パラメータの異なる

シミュレーションプログラムなどが考えられる。ここで想定しているバッチプログラムは、たとえば

```
napsack -f condition1.d > Result1
napsack -f condition2.d > Result2
```

のようにその内容がシェルスクリプトとしても利用可能なフォーマットで、各行の内容が「タスク」1つに相当する。タスクは実行プログラム1つとその計算結果をリダイレクトするためのファイルで構成される。ただし、タスクとして使用する実行プログラムは、同機種間の分散システムであればカレントディレクトリに置かれていればよいが、機種により実行形式が異なる異機種間の分散システムであれば、そのホストごとにパスの設定されたディレクトリに置かれている必要がある。同じ内容の実行プログラムがネットワーク上の異なる機種の数だけ必要となるが、これは異種機間の分散システムの場合はやむを得ない前提と考えられる。

実行プログラムのソースファイルが存在するのなら、タスクを転送する際にそのソースファイルが置かれているディレクトリ名も付加し、再コンパイルさせてから実行させるという方式も考えられる。しかし、タスクによってはその実行時間よりコンパイルにかかる時間の方が長いことも考えられ、再コンパイルさせてまでそのホストで実行させるかを判断するのは難しい。一般にタスクの実行時間を実行前に推定することはプログラムの解析などが必要であり負荷分散方式とは別の議論となるので、コンパイルに関しては本方式では採用していない。

2.3 ホストの負荷

N台のホスト $H_1, H_2, \cdots, H_N$がバス型通信ネットワークで結合されている分散システムを考える。前述したように各ホストには Network Task Distributor(NTD) というプロセスが1つ起動されており、本方式を使用するユーザはタスクを NTD に対して投入する。H_i上の NTD をNTD_iと書くことにする。

負荷分散アルゴリズムを考える場合、ホストの負荷を表す指標が必要である。簡単な負荷の表現方法としてよく用いられるものに、そのホスト上のタスク数 (あるいはプロセス数) がある [2][3]。

本方式ではホストの負荷として、そのホスト上に起動された NTD が扱うタスク数と、そのホストの利用状況を表す CPU のロードアベレージをパラメータとするものを採用した [1]。ここでいうタスク数とは、NTD 上にあるタスクの数を指し、そのホストで実行されているプロセス数を意味するのではない。このため、NTD 上のタスク数だけを負荷の指標とした場合、たとえば他のユーザが長時間の計算が必要な重いプロセスを実行しているときなど、正確に負荷状況を判断することはできない。つまり、同じ数だけタスクをもつホストでも、ロードアベレージが1のものと2のものとではタスクの応答時間が2倍異なることが予想できる。

ロードアベレージは、X ウィンドウアプリケーションの xload でよく知られるようにホストの負荷状況を示す簡単な指標として定着しており、本方式では NTD 上にあるタスク以外、すなわち実行状態にあるプロセスの CPU や資源の利用度を調べ、これをホストの負荷として考慮するために使用している。

本方式では、得られたロードアベレージの値を単純に他のユーザによるタスク数と考え、ホストの負荷 L を NTD 上のタスク数とロードアベレージ値の和として表現する。たとえば、ロードアベレージの値が 1.0 であれば他のユーザからのタスク数は1とみなし、そのときの NTD 上のタスク数が3であればそのホストの負荷は 4.0 と計算できる。一般にロードアベレージの値は変動が大きくもともと整数で与えられることは少ないが、その場合でも同様に和として計算するものとし、何らかの手段による整数化はしない方針とした。

ホストH_iの負荷をL_iとして表すと、L_iは次式で計算できる。

$$L_i = W\_Load_i \times (Num\_of\_Tasks_i + Load\_Average_i)$$

ここで、$Num_of_Tasks_i$はNTD_i上のタスク数で、$Load_Average_i$はH_iのロードアベレージである。また、W_Load_iはH_iの負荷の重み係数で、あらかじめ各ホストについて設定しておくものとする。これを用いるのは、本方式が異機種間の分散システムにおける負荷分散も想定しているためで、各ホストが搭載する CPU の処理能力の違いを考慮して負荷の重み付けを行っている。ホストごとにあらかじめ負荷の重み係数W_Loadを用意しておくことになるが、ホストの中で基準となるものを決め、そのW_Loadを1とし、他のホストのW_Loadは何らかのベンチマークテストを用いた結果から、その基準ホストの結果に対する計算時間の比で決定する。ベンチマークテストの種類によって評価が分かれ、ホストごとにW_Loadを厳密に設定することはできないが、異機種間の処理速度の違いを表現する目安として使用する。W_Loadは処理速度の速いホストほど値が小さくなるように設定され、ホストの負荷 L は処理速度の速いホストほど小さい値として計算されることになる。

また、計算したL_iがあらかじめ定めた閾値 $Threshold$ を越えるかどうかで決まる、ホストH_iの (負荷) 状態 $state_i$を導入する。ホストの状態を導入するという手法は過去に提案された動的負荷分散方式にはよく用いられており [2][4]、本方式でも採用している。

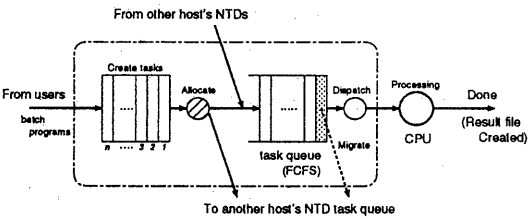

図 1: タスクの生成から実行終了まで

ここではホストの状態として、*heavy*と*light*の二つを設定し、次の式で定義する。

$$state_i = \begin{cases} light & (L_i < Threshold) \\ heavy & (L_i \geq Threshold) \end{cases}$$

light 状態には NTD 上のタスクがある場合とない場合とがあるが、後者を便宜上 *idle* 状態と呼ぶことにする。

ホスト状態が *heavy* のときタスクが余っていると考えることができ、余分なタスクを他の *light* 状態のホストへ転送し、処理を依頼することにより負荷の分散が行われる。

3 負荷分散アルゴリズム

負荷分散の対象となるタスクには、

生成タスク あるホストで、ユーザにより新たに NTD に依頼されたタスク

待ちタスク ユーザに依頼されてもすぐに実行されず、あるホストの NTD 内で待ち状態にあるタスク

の二種類がある。

両者とも分散させるタイミングも分散させる方針も異なる。本論文中では、生成タスクを分散させることを「割り当てる (allocation)」といい、待ちタスクを分散させることを「移送する (migration)」ということにする。

各ホストにおけるタスクの流れを示したのが図1である。ユーザから NTD へ投入されたタスクは、まず後述する生成タスク割当てアルゴリズムにより、そのホストの負荷状況に応じて、自ホストか他のホストかへの割当てが決定され、割当て先の NTD 内部のタスク待ち行列に入る。もし待ち行列が空の場合、そのタスクは即座にプロセスとして実行される。タスク待ち行列は FCFS(First Come First Service) 方式で、1 タスクの実行が完了するたびに、先頭にあるタスクから順にプロセスとしてディスパッチされる。また、先頭にあるタスクは後述する待ちタスク移送アルゴリズムにより他のホストへ移送されることもある。本方式では 1 ホストの NTD が同時に実行可能なタスクは 1 つと設定した。

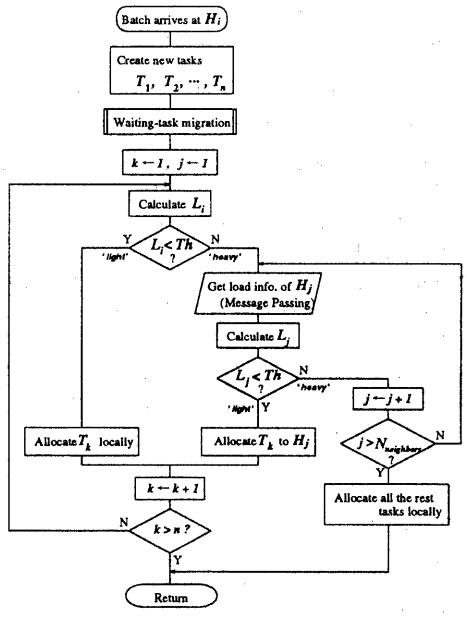

図 2: 生成タスク割当てアルゴリズム

3.1 生成タスク割当てアルゴリズム

このアルゴリズムは新たにタスクが生成されたときに実行されるもので、その流れ図を図 2に示す。ユーザから n 個の独立なタスクを含むバッチプログラムが NTD に投入されると、

1. まず、バッチプログラムは n 個の独立なタスク $T_1, T_2, \cdots, T_n$に分解され、割当てアルゴリズム (詳細は 3.3で述べる) により各タスクを割り当てるホストが決定される。

2. 自ホストへの割当てが決まったタスクは待ち行列に入り、プロセスとしてディスパッチされるのを待つ。ディスパッチ後、実行が終了すれば結果ファイルを生成し、そのタスクの処理が完了する。

3. 他のホストへの割当てが決まったタスクはメッセージとして組み立てられ、メッセージパッシングにより転送される。分散ファイルシステムを前提としているので、実際にはタスク本体ではなくその識別子 (タスクの実行プログラム名と結果出力ファイル名など) が送られる。

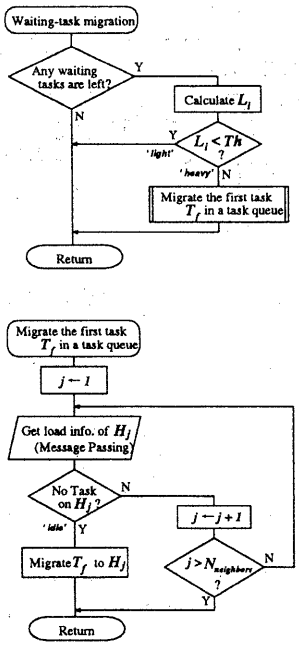

図 3: 待ちタスク移送アルゴリズム

3.2 待ちタスク移送アルゴリズム

待ちタスクを移送させるアルゴリズムを図 3に示す。このアルゴリズムは以下の二つの条件のいずれかを満たしたときに実行に移される。

1. 新しいタスクが生成されたとき、タスク待ち行列が空でなければ、まず待ちタスクの移送を考える。その後で生成タスクの割当てを行う。
2. タイムカウンタを用意し、最後の負荷分散アルゴリズムの起動から一定時間 T_{dist} が過ぎたときに待ちタスクの移送を考える。その結果、タスク移送が行われたかどうかに関係なくタイムカウンタはリセットされる。ただし、上記 1で起動した後にもタイムカウンタをリセットする。

いずれの場合にしても、一度の待ちタスク移送アルゴリズムの起動で移送可能な実行待ちタスクは、待ち行列の先頭にある一つだけとした。

3.3 動作の詳細

負荷分散アルゴリズムで使用するメッセージは以下の 4 種類である。

$Request\_Load\_Info_{ij}$ H_iがH_jの負荷情報（ロードアベレージと NTD 上のタスク数）を知りたいときに、H_iからH_jへ送られるメッセージ。

$Load\_Info_{ji}$ H_iから送られた $Request\_Load\_Info_{ij}$ を受け取ったH_jが自分の負荷情報をメッセージとして組み立てたもので、H_jからH_iへ送られる。

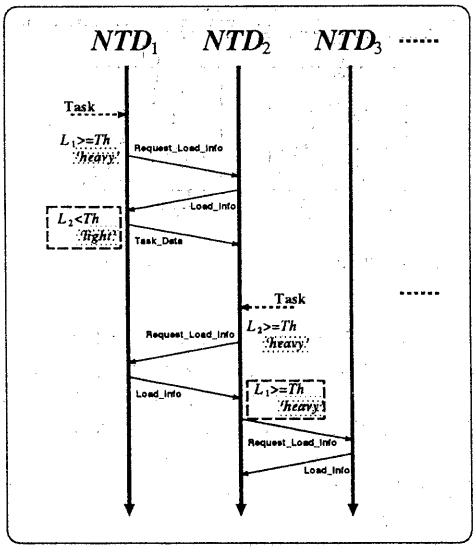

図 4: ホスト (NTD) 間のメッセージ通信

$Task\_Data_{ij}$ H_iからH_jへ転送されたタスクの識別子を含むメッセージ。

$Task\_Termination_{ji}$ H_j上でH_iから転送されたタスクの実行が終わったときに、H_jがH_iに送るメッセージ。

生成タスク割当てアルゴリズムの実行時に、各ホスト上の NTD 間でメッセージ通信が行われたときの様子を示したのが図 4である。タスクの生成時に待ちタスクがあれば、その生成タスクの割当てを決める前に待ちタスクの移送を考えなければならないが、図を簡潔にするために省略した。

本負荷分散方式では、各ホストにはそのネットワーク上で識別可能な ID(番号) が与えられ、任意の 2 つのホストH_i, H_jに対して、$i < j$であれば $W_Load_i \leq W_Load_j$と設定している (等号はH_i, H_jが同一機種のとき)。すなわち、全ホストは、処理速度の速いもの順に順序付けられている。

ホストが *heavy*状態であれば、そのホストの NTD はタスクの分散を行う。その際、分散先となるホストとして、まず ID の小さいホストを選んで$Request_Load_Info$ メッセージを送信し、そのホストから$Load_Info$ メッセージが返信されるのを待つ。

図中の NTD_1のように 1 回の負荷情報の要求でタスク 1つの割当てが決まる場合もあるが、NTD_2のように 1 回の負荷情報の要求だけでは決まらない場合も考えられる。最悪のケースで (全ホスト数) − 1 回の$Request_Load_Info$ メッセージが送られ、全ホストが *heavy*状態とわかれば、割当ての対象となっていたタスクは自ホストの待ち行列に入る。

ここで、メッセージ通信の回数を極力減らすた

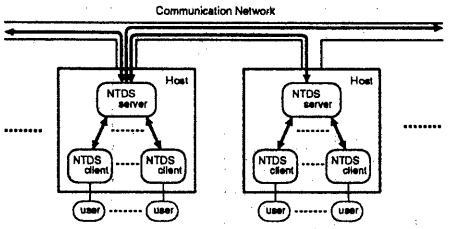

図 5: NTDS サーバと NTDS クライアント

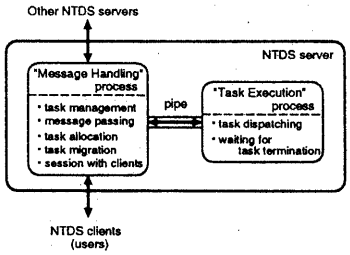

図 6: NTDS サーバの構造と機能

め、一度の負荷分散アルゴリズムの起動につき、他のホストへ負荷情報を要求する回数を他のホスト1つにつき1回と制限した。他のホスト1つにつき1回の負荷情報の獲得では、タスクを多く含んだバッチプログラムが投入された場合、不正確な負荷情報に基づいてタスクの割当てを行ってしまう可能性がある。このため、あるホストが *heavy* 状態となりその NTD が他のホストへのタスク分散を行うとき、他のすべてのホスト上の NTD は、そのタスク分散が完了するのを待つこととする。このためには NTD 間の条件同期が必要になるが、ここではロックファイルの作成により、これを実現している。

4 インプリメンテーション

前章で述べた負荷分散アルゴリズムに基づき、ネットワークタスク分散システム (Network Task Distribution System, NTDS) を実際の分散システム (UNIX ネットワーク) に実装した。分散システムへの実装の作業には、BSD 系 UNIX を OS として搭載したワークステーション上で C 言語によりコーディングを行った。

ネットワークタスク分散システム (NTDS) は、クライアントサーバモデルとして構築され、タスクを管理する NTDS サーバと、それに対してタスクを投入する NTDS クライアントに分かれる。NTDS サーバは2章で述べた NTD プロセスに相当し、また NTDS クライアントは本方式を使用するユーザに相当する。各ホストに起動される NTDS サーバは1つだけだが、NTDS クライアントは複数存在してもかまわない (図 5)。

図 6に示すように、NTDS サーバは、メッセージを処理する「メッセージ処理プロセス」とタスクの実行を制御する「タスク実行プロセス」という互いにパイプで接続された2つのプロセスで構成される。

メッセージ処理プロセスは、複数の NTDS クライアントや他のホストの NTDS サーバとソケットを通じたデータグラム型のメッセージ通信を行い、ユーザ (NTDS クライアント) からのタスクの投入を待ったり、他のホスト上の NTDS サーバの負荷情報を求めるメッセージを送るなどして負荷分散アルゴリズムによりタスクの割当てや移送を行う。またパイプを通じてタスク実行プロセスに、実行するタスクの識別子を送りそのタスクのディスパッチを依頼する。

タスク実行プロセスは、メッセージ処理プロセスからのタスク実行の依頼がくると fork してそのタスクを子プロセスとして実行し、実行が終了すればパイプを通じてメッセージ処理プロセスに伝え、次のタスク実行の依頼を待つ。

ユーザはタスクを投入する際、NTDS クライアントから NTDS サーバに接続要求を出す。NTDS サーバから接続応答のメッセージを受け取るとタスクを投入可能な状態となる。

5 性能評価

UNIX ネットワーク上に実装した NTDS の性能評価実験について述べる。実験の目的は、NTDS を使用することにより負荷分散を行わない場合に比べてシステム全体の性能がどのくらい向上したかを測定することである。性能の指標は、タスクの平均応答時間の短縮率とする。

使用した UNIX ネットワークは、SUN SPARCstation IPX, SUN SPARCstation2 GS が各1台、そして SONY NEWS NWS-3460 が2台の合計4台の UNIX ワークステーションをイーサネットで接続した異機種間の分散システムである。説明のため上記の UNIX ワークステーションを順に S_1, S_2, R_1, R_2 と呼ぶことにする。

評価用のタスクとして、単純な浮動小数演算を N_l 回繰り返し、実行に費やされた実時間を出力するプログラムを作成した。このプログラムで N_l が 50×10^6 のものを実行させた結果を表1に示す。このプログラムで出力される時間は、CPU を必要とした時間 (CPU 時間) ではなく、実行を開始した時刻から終了した時刻までの時間を計測してい

表 1: 評価用タスク (N_l=50×10^6) の平均実行時間

ホスト	S_1, S_2	R_1, R_2
平均実行時間 (秒)	43	50

るので、NTDS を用いない場合には直接タスクの応答時間が得られることになる。たとえば、同じループ回数のタスク 2 つが同時にプロセスとして実行されれば、両タスクとも 1 つだけ実行された場合に比べて 2 倍の応答時間がかかり、平均応答時間も 2 倍となる。

一方、NTDS を用いる場合、タスクがプロセスとしてディスパッチされるまでの NTDS 内の待ち時間があるため、このプログラムで出力される時間をそのまま応答時間として考えることはできない。NTDS を使用した実験の場合、タスクの応答時間は NTDS クライアント側で計測するターンアラウンド時間としている。

以上のような評価用タスクを 1 つ含み、Bourne シェルのシェルスクリプトとしても使用可能なバッチプログラムを生成させる。バッチプログラムの到着間隔は平均λ秒の指数分布に従い、バッチプログラム中のタスクのループ回数は平均N_l回の指数分布に従うものとし、両者とも各ホストについて独立である。このようなバッチプログラムを各ホストに独立に生成させる。

ホストの負荷 L の計算に関しては、ホストの状態を決定する *Threshold* を 1.0, ホストの負荷の重み係数*W_Load*は S_1, S_2を 0.8, R_1, R_2を 1.0 とした。また、NTDS で待ちタスク移送アルゴリズムを起動する間隔 T_{dist}は 30 秒とした。

5.1 結果と考察

図 7は、タスクの平均ループ回数N_lを 50×10^6とし、バッチプログラムの平均到着間隔λを変化させたときの実験結果である。

まず、負荷分散を行わなかった場合の結果について考察する。仮に各ホストですべてのタスクが前のタスク (プロセス) の終了後に到着した場合、表 1より、負荷分散を行わない場合のタスクの平均応答時間は $(43\times2+50\times2)/4 = 46.5$ 秒程度となることが推定できる (この値をT_rとする)。しかし実際にはタスクの到着間隔や実行にかかる時間は一定ではないので、同時に 2 つ以上のタスク (プロセス) が並行に実行されることが多いので、その分応答時間が悪化してしまう。このことは実験結果にも現れているが、とくに、タスクの到着間隔を小さくしてシステムに与える負荷を大きくするほど、応答時間の増加が著しくなっている。グラフよりλがT_rのときでも、負荷分散を行わないとタスク (プロセス) の平均応答時間は約 $2T_r$かかることわかる。

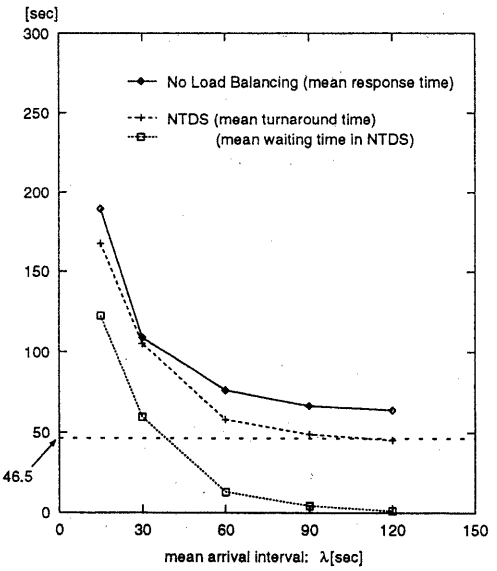

図 7: タスクの平均応答時間 (1) (N_l= 50×10^6 [回])

次に、NTDS を使用した場合に注目する。全体的に眺めるとグラフは負荷分散を行わない場合とほぼ同じ特性を示しているが、平均応答時間は負荷分散を行わないときよりも短縮されているのがわかる。1 ホストの NTDS サーバは同時にタスクを 1 つしかプロセスとしてディスパッチしないため、他のユーザによる重いプロセスがない場合には、すべてのタスクの実行時間はほぼ一致することが予想される。したがってタスクの平均応答時間は、タスクが実行されるまでの NTDS 内の待ち時間の大きさに依存する。λがT_rと等しくなる付近から大きくなると、負荷分散を行わないときに比べてタスクの平均応答時間は平均待ち時間の短縮に伴い急速にT_rに近づいている。

平均応答時間でいうと、λが 60 のとき負荷分散なしの場合が約 $1.7T_r$であるのに対し、NTDS を使用した場合では約 $1.3T_r$に短縮されており、λが 90 のときで負荷分散なしの場合が約 $1.4T_r$であるのに対し NTDS を使用した場合ではほぼT_rに一致している。さらに、λが 120 になると負荷分散なしの場合で約 $1.4T_r$だが、NTDS を使用した場合では平均待ち時間が 1 秒程度で平均応答時間はT_rよりも幾分か小さくなっている。これは、余分なタスクが*W_Load*の値が小さく設定された処理能力の高いホストに集まるためで、異機種間の負荷分散が効果的に行われていることがわかる。

以上の結果をまとめると、NTDS を利用すれば負荷分散を行わない場合と比べてタスクの平均応答時間が短縮され、システムに適度な負荷が与えられた場合、NTDS の負荷分散効果が顕著に現れ

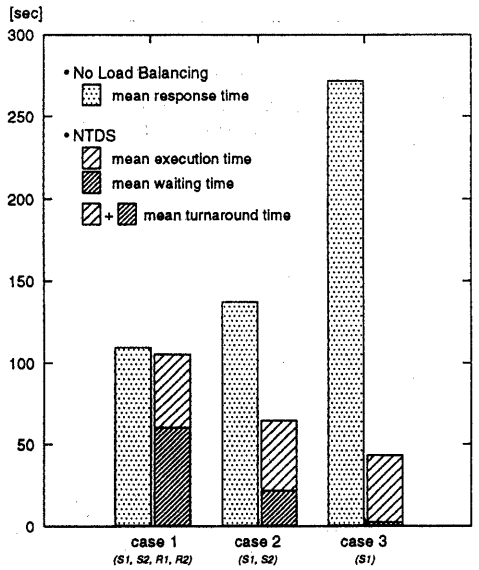

図 8: タスクの平均応答時間 (2) (N_l= 50×10^6 [回], λ= 30 [sec])

てくるということである。

次に、NTDS が空いているホストを効果的に利用しているかを調べるために、生成させるバッチプログラム数をホストごとに変化させた実験結果を図 8に示す。この実験では、合計 4 台のホストに

case 1. S_1, S_2, R_1, R_2に各 10 個ずつ

case 2. S_1, S_2に各 20 個ずつ

case 3. S_1のみに 40 個

のバッチプログラム (タスク) を生成させた。バッチプログラムの平均到着間隔λは 30 秒とし、タスクの平均ループ回数N_lは先の実験と等しく 50×10^6 と設定している。参考のため負荷分散なしの場合の実験も行った。

この図から、NTDS を使用したとき、各ホストに一様にタスクを生成させた場合に比べて生成させるホストを限定した場合の方がタスクの平均応答時間が小さくなっていることがわかる。タスクの平均到着間隔λを 30 秒に設定したため、各ホストに単位時間当りに到着するタスク数の平均は各ホストでほぼ等しいが、生成させるホスト数を多くすると単位時間当りのシステム全体のタスク数もその分増加するということを考えれば、各ホストに一様にタスクを生成させた場合の平均応答時間が一番悪くなるという結果が得られるのは当然といえる。

しかし負荷分散を行わない場合の結果と比較すれば、ホスト間で負荷の偏りがあった場合に NTDS が余分なタスクを空いてるホストへ効果的に分散させることにより平均応答時間を大幅に改善させていることがわかる。

6 まとめ

本研究では、分散システムにおけるタスクの平均応答時間の短縮を目的とした、異機種間の分散システムにも適用可能な動的負荷分散方式を提案した。さらにこの負荷分散方式をネットワークタスク分散システム NTDS として、NFS を利用した異機種間の UNIX ネットワーク上にクライアントサーバモデルを用いて実現した。

実際の使用を想定した人工負荷による実験の結果から、一部のホストにタスクが集中した場合、余分なタスクを効果的に空いているホストに分散させることで、負荷分散を行わないときよりもタスクの平均応答時間が短縮されることが認められ、本システムによる負荷分散効果が確認された。

今後の課題としては、まずタスクの粒度の変化や本方式に関する種々のパラメータ (待ちタスク移送アルゴリズムの起動間隔 T_{dist}や $Threshold$ など) 設定など、様々な条件による本システムについての幅広い性能評価が挙げられる。また、優先度付きのタスクを導入した場合のスケジューリング方式の見直しなども挙げられる。

参考文献

[1] 吉田, 高井, 佐藤: "UNIX ネットワークにおけるバッチ型タスク分散システム", 電子情報通信学会 1993 年秋季大会講演論文集, Vol.6, p.87, Sep. 1993.

[2] Tony T. Y. Suen and Johnny S. K. Wong: "Efficient Task Migration Algorithm for Distributed Systems", *IEEE Trans. on Parallel and Distributed Syst.*, Vol. 3, No. 4, pp.488–499, July 1992.

[3] 山井, 下條, 宮原: "同報通信機能をもつ分散システムにおける負荷分散アルゴリズム", 電子情報通信学会論文誌 D–I, Vol. J75–D–I, No.8, pp.536–544, Aug. 1992.

[4] Thomas Kunz: "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme", *IEEE Trans. on Software Eng.*, Vol. 17, No. 7, pp.725–730, July 1991.

[5] N. G. Shivaratri, P. Krueger, and M. Singhal: "Load Distributing for Locally Distributed Systems", *Computer*, Vol. 25, No. 12, pp.33–44, Dec. 1992.