資源情報流通サーバ SSS-Server を用いた負荷分散

佐々木盛朗　　亀沢寛之　　松本尚　　平木敬

東京大学 大学院理学系研究科 情報科学専攻
〒 113 東京都文京区本郷 7–3–1
Email: {shigero, kame, tm, hiraki}@is.s.u-tokyo.ac.jp

あらまし　分散処理において、ネットワーク上の資源の有効な活用は重要な課題であり、資源を有効に用いて効率良くアプリケーションを実行するにはネットワーク資源の状態を正しく把握をする必要がある。従って、実用的コストで正確なネットワークの資源情報を集めることが効率の良い並列分散実行の鍵である。SSS-Server は、ネットワークで結合されたシステム上での情報収集を行ない、負荷情報の取得を支援する。本稿では、 SSS-Server を使ったアプリケーションとして、依存関係のあるジョブをクライアントに投げていくサーバ SDA(Server of Directed Acyclic graph) を作成し、負荷分散における SSS-Server の有効性を評価した。

キーワード　SSS-Server,SDA

Load balancing with SSS-Server

Shigero SASAKI　　Hiroyuki KAMESAWA　　Takashi MATSUMOTO　　Kei HIRAKI

Department of Information Science, Faculty of Science, University of Tokyo
3-7-1 Hongo Bunkyo-ku Tokyo 113, Japan
Email: {shigero, kame, tm, hiraki}@is.s.u-tokyo.ac.jp

Abstract　In distirbuted processing,it is important that all resources on network is made the most of. We need to grasp a state of resources on network for executing applications effectively.So the key for effective parallel distributed execution is exact and inexpensive practically information of a network. SSS-Server offers information on the network system, and supports getting load information. In this study,SDA which distributes jobs with dependancy proposed, and the effect of SSS-Server was evaluated.

key words　SSS-Server,SDA

1 はじめに

分散処理において、ネットワーク上の使われていないプロセッサ、メモリなどの資源の有効な活用は重要な課題である。資源を有効に用いて効率良くアプリケーションを実行するには、ネットワーク資源の状態を正しく把握をする必要がある。従って、実用的コストで正確なネットワークの資源情報を集めることが効率の良い並列分散実行の鍵である。しかし、それぞれの資源に個別に問い合わせをすると、状態の把握にコストがかかるか、不正確な情報しか得られない。

現在、ワークステーションは様々な場所でつかわれており、LAN でつながれ、NOW(Network Of Workstation) として使われる。このような使用形態をとる場合、ネットワーク上の資源情報の取得は重要な意味を持つ。例えば、あるワークステーションは 10 個のジョブを持ち、あるワークステーションはジョブが 1 つもない、といった状況を考える。この時、要求に答えられないほどのジョブを持つワークステーションと何もしていないワークステーションが同時に存在し、非効率的である。実際、多くのジョブを持つワークステーションと、少ないジョブしか持たないワークステーションがある状況はしばしば現れ、効率的にワークステーションを使用し、スループットを上げるためには、各ワークステーションの負荷を平均化することが必要である。そのためには、負荷情報を何らかの形で取得しなければならない。

東京大学理学部平木研究室で開発した SSS-Server [1] は、ネットワークで結合されたシステム上で、負荷情報、メモリの使用状況、端末情報等の情報収集を行なうサーバである。SSS-Server は、既存の汎用オペレーティングシステム (以下 OS) 上で動き、さまざまな資源情報を自動的に効率良く集め、低コストで確度の高い情報の提供を行なう。また、デフォルトで集める情報以外に、集める情報をクライアントから指定できるという機能によって汎用性のあるサービスを提供している。SSS-Server が提供する情報があれば、分散環境において有効に負荷分散を行なうこともできる。負荷情報を利用すれば、例えば、使用するワークステーションをユーザに意識させずに、自動的に負荷分散を行なうようなアプリケーションの開発も可能である。

本稿では、SSS-Server を使ったアプリケーションとして、SDA(Server of Directed Acyclic graph) を作成した。SDA は、依存関係のあるジョブをクライアントに分配するサーバであり、同期を取りつつ負荷分散を行なう。負荷分散を行なうにあたって、SDA は負荷情報を取得する必要があり、その正確さ、安価さによって実行速度に差がつくため、負荷分散における SSS-Server の有効性を評価することができる。

2 SSS-Server

この節では、SSS-Server が作られた背景と、その基本動作について述べる。

2.1 SSS-CORE

汎用超並列オペレーティングシステム **SSS-CORE**[2] は, 並列アプリケーションと協調動作することで、効率を極力落すことなくマルチユーザ / マルチジョブの汎用環境を実現する、分散メモリ型並列計算機およびワークステーションクラスタ環境を対象とした汎用オペレーティングシステム (汎用 OS) である。SSS-CORE はシステムの資源管理に階層性を導入して資源管理の効率化を行うことにより、スケーラビリティつまり超並列超分散計算環境に対応している。

SSS-CORE では Snoopy Spin wait(**SS-wait**)[4] と呼ばれる資源割当状況によるスケジュールオプションつきのスピンウェイトを同期の基本としている。さらに、高速なユーザ通信 / ユーザ同期を提供する非対称分散共有メモリ (Asymmetric Distributed Shared Memory: ADSM) を指向している。そして、ADSM を実現するためのメモリベース通信機能 [3] の一部が、既に実装されている。

NUMA 型の並列計算機システムでは、メモリアクセスコスト及び通信コストが、距離に影響されるため [5]、これらのコストが小さく抑えられるようなスケジューリングが必要である。そこで、SSS-CORE では、計算機のインターコネクションネットワークの階層構造と一致した構造の資源管理構造を用意して資源情報の管理をし、各プロセスはスケジューリングに対して制約を提示する。カーネルレベル・スケジューラは資源情報と、プロセスから提示されたスケジューリング制約にしたがって、プロセス単位に資源の割当を行なう。そうすれば、資源情報を利用することで、タイムスライス毎の資源の割り当て状況をスケジューリングに反映することが可能となる。更に、提示されたスケジューリング制約に従ってスケジューリングすることで、プロセス毎に異なる資源への欲求を満たすことができる。資源管理構造は並列計算機の構造と同じであるために、割り当てを行なうプロセッサ間の通信コストを把握したうえでのスケジューリングが可能である。

効率的なスケジューリングを実現するために、SSS-CORE の情報収集機構は、

1. 非ローカルの資源情報の安価な提供
2. eagar な情報収集よるユーザアプリケーションへのレスポンス向上
3. 階層的な情報流通機構による効率が良い情報収集

という特徴を持っている。 1 は情報収集のオーバヘッドを抑えるため、 2 は正確な情報を集めるため、 3 はスケーラビリティを実現するためのものである。

2.2 SSS-Server の基本動作

SSS-Server は、汎用 OS 上で単体のソフトウェアとして、 2.1 節で述べた SSS-CORE の情報収集機構の特徴を実現するものである。さらに、ユーザに自由な流通データ作成を許すことで汎用性を向上させている。 SSS-Server は、

- あらかじめ集めておいたネットワーク情報をユーザに低コストで開示
- ユーザによる自由な流通データの作成
- パケットの整理によるネットワークの負荷の抑制

といった特徴を持つネットワーク情報流通サーバであり、 SSS-CORE の部分機能を NOW(Network of Workstations) に実装したものである。

SSS-Server は、同一ネットワーク内の他のサーバと互いに排他制御をしながら、ローカルなホスト情報を自分の属するサブネットワークにブロードキャストする。データパケットを受けとった SSS-Server はそれを共有メモリ上に格納する。その際、パケットの再送などによる混乱を避けるために、論理クロック [6] を用いている。これは、ネットワーク上の各イベントに対して唯一のタイムスタンプを与える。そして、各サーバの変数 clock とパケットに押される timestamp によって、ネットワークに単一のクロックを作る。 SSS-Server は、 clock と timestamp を次のように利用する。

1. timestamp $<$ clock ならば、パケットを棄却
2. timestamp = clock かつ、パケットの送出元が直前のパケットと同じであるならば、受理
3. timestamp $>$ clock なら受理、 clock = timestamp とする
4. timestamp = 0 なら受理

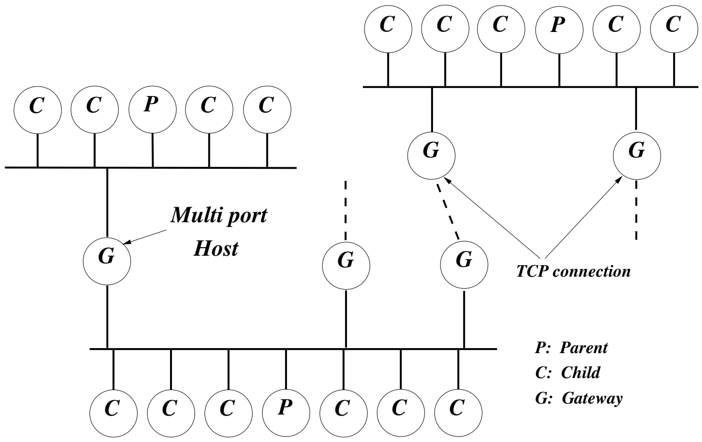

図 1: SSS-Server の階層構造

2 は、複数のパケットを送る時は、それらは全て同じタイムスタンプを押されるため、これらの処理のためのものである。 SSS-Server は、この論理クロックによって、階層内での時間的整合性を保つ。

サブネットワークを越える通信は TCP を用いて行なわれ、得られた外部ネットワークの情報はローカルネットワーク内にブロードキャストされる。 SSS-Server はコンフィグ情報及び NIS(The Network Information Service) を用いてネットワークに疑似的な階層性を与えてパケットの流通経路を形成する。この流通経路は、アルゴリズム的に一つのサーバが同一内容のパケットを 2 度以上受けとらないことを保証する。

最初に起動される SSS-Server は、各ホストの名前とネットワークアドレスを書いたコンフィグファイルを読み込み、ネットワーク構造を構成する。コンフィグにないホストからのアクセスは受け付けない。また、異常が発生した際には、柔軟にネットワーク構造を再構成してネットワーク構造を維持する。 SSS-Server はユーザに対して自由な情報の作成を許している。ユーザはデータを作成し SSS-Server に渡すことで、サーバの動いているネットワーク全体でそのデータを受けとることができる。

2.3 関連研究

現在、ネットワーク上のホスト情報を流通させるシステムとしては、 rwhod とその改良版である ewhod が広く用いられている。

rwhod は、現在の各ユーザのログイン名、端末名、およびログイン時刻等の who で得られる情報を全てのマシンに対して作成する rwho、現在の時刻、システム稼働時間、及び最後の 1 分、 5 分、 15 分間の実行待行列の平均ジョブ数といった uptime で得られる情報を全てのマシンに対して作成する ruptime

が使用するデータベースを保守するサーバである。rwhod は、ステータス情報の送受信を行う。送信する時は、定期的にシステムの状態を調べ、ステータス・メッセージを作成し、ネットワーク上にブロードキャストする。受信する時は、他の rwhod サーバのステータス・メッセージの受信を監視し、その妥当性を検査し、ディレクトリ /var/spool/rwho にある一連のファイルに記録する。ステータス・メッセージは、ほぼ 60 秒ごとに 1 回生成される。

rwhod はせいぜい 10 台ぐらいのマシンが 1 つのネットワークに繋がっている時にはさほど負担にもならないが、数十台にもなると、かなりな network eater,CPU eater になり得る。また、 2 つ以上のネットワークがゲートウェイを介して接続されているような場合、相互に情報を交換するという機能がない。ewhod は、 NFS(The Network File System) のクライアント (特に diskless) の負担を軽減できる、他のネットワークとも情報を交換することができる、情報の流量をある程度制御できる、という点において rwhod を拡張したもので、パケットの形態は全く同じで、ネットワーク内に両者が共存可能ある。しかし、これらのサーバは、ユーザによる自由な流通データの作成を許していない。また、今までの比較的遅いネットワーク (10Base-T) 上では正常に動作していたが、高速なネットワーク (100Base-T) 上ではうまく動作しない。これから一層高速になっていくネットワークに対応できないのは問題であり、その原因は以下の 3 つである。

1. ネットワークを越えるパケットの制限のない送受信
2. データの圧縮・パケットのコンパクト化を行なわないことによる多大なパケット量
3. 排他制御を行なわない、一定の時間経過によるパケットの送受信

SSS-Server は上記 3 点を踏まえて設計されており、高速ネットワーク上でも十分その機能を果たす。また、 2.2 節で述べたように、ユーザによるデータの作成が可能なことも、 SSS-Server の優れた点である。

3 SDA

SDA は、 SSS-Server を利用し、柔軟で効率的な分散処理を実現するサーバである。 SDA は、与えられた依存関係のあるジョブを、 NOW 上で複数のクライアントに動的に分散させて処理する。ジョブの依存関係は DAG で表せるため、 SDA は図 3のようなジョブの DAG を表したテキストファイルを解釈し、

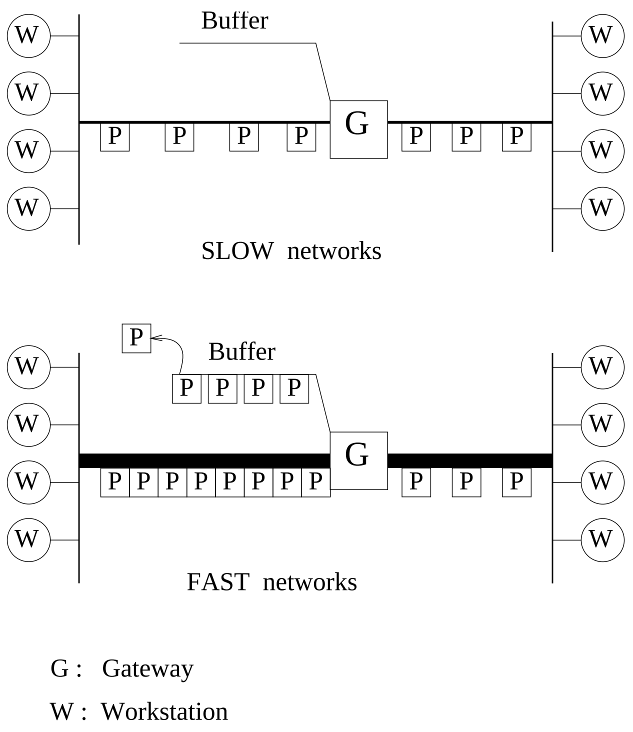

図 2: パケットの欠落

他のワークステーションの負荷情報を見て、ジョブを効率よく分散させる。

依存関係のあるジョブを処理するのは、ユーザが依頼するジョブは 1 つ 1 つが独立したものであるとは限らないためである。例えば、 make をユーザが実行したとすると、それぞれのソースをコンパイルした後リンクする、というように、ジョブに依存関係が生じる。図 3の長方形はサーバに投げるジョブを表しており、矢印で指されているジョブは、指しているジョブが終了するまで実行されない。矢印で指されていないジョブは実行可能であり、ジョブが複数ある場合は、それらは並列に実行できる。中に job 100 と書かれた長方形があるが、これは job 1 の 100 倍の時間が終了までにかかるジョブを表している。必要なのは終了までにかかる時間のジョブ間の比なので、 job 1 が終了するまでにかかる時間は任意でよい。それぞれのジョブが終了するまでの時間は異なり、その時間はあらかじめ分かるとは限らないが、ここでは全て実行前に分かっているものとして扱う。

なぜなら、ジョブの重さが確定できないければ負荷分散が行なえず、また、重さをを何らかの形で見積もった場合、その評価方法が実行時間に大きな影響を及ぼすためである。あらかじめ全てのジョブの重さが決定されているのならば、静的にジョブを実行するワークステーションが決定されてしまうが、負荷が動的に変化する環境を想定しているので、クライアント

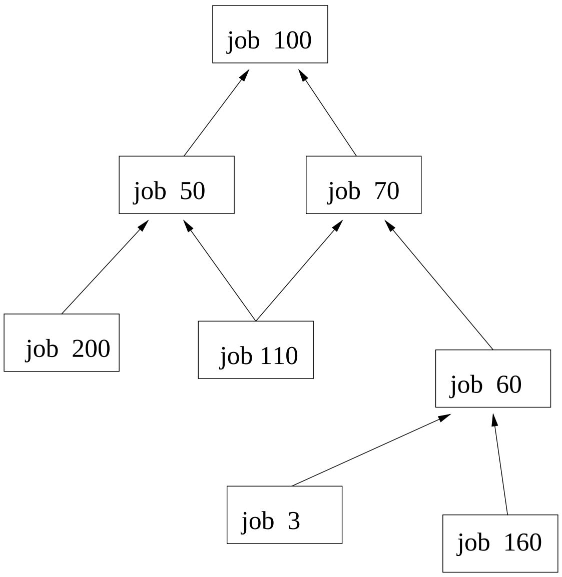

図 3: ジョブの構造

はサーバが管理していないジョブも投げられるため、そのようなことはない。

通信プロトコルには、 UDP に着信と到着順序を保証する機構を付加したものを用いた。 TCP を使わないのは、ジョブの転送にともなう通信オーバーヘッドを減少させ、 1 つのポートで多くのホストと通信するためである。

3.1 パーザ

パーザが解釈するテキストファイルは、依存関係が書かれた行とジョブの内容が書かれた行が交互に並んでいる。依存関係が書かれた行は、 0:1 2 3 のようになっていて、コロンの前には次の行に書かれているジョブを持つノードの名前が書かれていて、コロンの後にはこのノードのジョブが依存しているジョブのノードの名前が書いてある。例を下に示す。

```
1:2 3
job 10
2:4
job 20
3:4
job 30
4:
job 40
```

3.2 通信方法

UDP による通信ではパケットの到着は保証されず、到着したとしても発信された順に到着しているとは限らない。そのため、パケットの到着を保証する必要がある。本稿では、多数のパケットを同時にやりとりしない限り、パケットの欠落等の問題は起こらないような環境を想定しているため、何か問題が生じた時の処理が重くなっても、問題が起こらない場合の通信は軽く行なえる方がよい。パケット欠落の問題を解決するため、以下のような方法を用いた。

1. 送り手は、メッセージに送り先別の連続した番号を振る
2. 受け手は、受け取り先別に、何番目のメッセージまで受け取ったかを表す番号を保持する
3. パケットの着信を保証する手段として、 Acknowledge を送る。

1 のようにすれば、受け手はパケットの欠落、到着順の乱れを、不連続な番号を受けとることにより検出する。また、再送要求に答えるため、到着が確認されるまではバッファにメッセージ保持する。バッファは有限であるため、定数個のメッセージが送られると、それらの着信が保証されるまで新たなメッセージを送ることはできない。

2 のように番号を保持していて、その次の番号を持つメッセージが届いたら、また次の番号のメッセージを待つだけでよい。しかし、その先の番号が振られたメッセージを受け取った場合、パケットが欠落したか誤った順序で到着したことになる。この場合、受け手はメッセージをバッファに保持し、間の番号を持つメッセージを受けとってから渡す。現在の番号より小さいか、バッファにあるメッセージと同じ番号を持つメッセージは、重複して届いたものなので無視する。

3 で触れた Acknowledge とは、どこまでメッセージが届いたかを送り手にしらせるものである。メッセージの送り手は、 Acknowledge を返すように要求するメッセージを送信する。受け手はこのメッセージを受け取ると、読み込み番号を書いて送り返す。読み込んだ番号を付け加えたものを Acknowledge として返す。 Acknowledge を受け取った送り手は、書かれている番号から再送の判断を行なう。

Acknowledge や Acknowledge 要求が届かない可能性があるが、送り手は、ユーザが指定した時間内にメッセージを 1 つも受け取らなければ、パケットが欠落したと判断し、必要な Acknowledge 要求を再度行なう。必要な Acknowledge 要求とは、前回の Acknowledge 要求からユーザが定義した時間だけ経過しているのに Acknowledge を受け取っていない通信相手への、または、 1 つも Acknowledge 要求は出していないが、今まで送ったメッセージの保証がない通信相手への Acknowledge 要求である。

3.3 負荷分散の方法

単純なタスクの遠隔実行は、性能の向上より、むしろ低下の原因となる。なぜならプロセスマイグレーションは、どんなに効率良く実装されようともオーバーヘッドになるからである [7]。本稿では以下の 3 つの方法で負荷分散を行なう。

1. ランダムにジョブを投げる方法
2. /dev/kmem から得た、 piggyback される Active Process の数を用いて求めたサーバの処理能力の比に並列実行可能なジョブを分割し、それらを投げる方法
3. SSS-Server を使って得た Active Process の数を用いて求めたサーバの処理能力の比に並列実行可能なジョブを分割し、それらを投げる方法

上の 3 つの方法を選んだ理由を述べる。 1 は、比較の基準にするために選んだ。 2 は、ユーザが個々のホストの情報取得のためにコードを書いて、独自に集めた情報に基づいて負荷分散を行なうと、 SSS-Server から得た情報に基づいて負荷分散を行なった場合に比べてどうなるかを見るためである。 2 で piggyback を行なったのは、 piggyback を行なわないと負荷情報を送るように要求するメッセージを送ることになるからである。負荷情報が返ってくるまでの時間のオーバヘッドを隠蔽するように負荷情報を要求するのは非常に困難であり、結局は定期的に要求することになる。これでは、 SSS-Server と同じことをやることになる。 piggyback のオーバヘッドは非常に小さいが、得られる情報の正確さは、相手からのメッセージの量による。これは SSS-Server とは異なる、広く行なわれている情報収集方針であり、 SSS-Server の効果を評価するための比較対象になる。

2,3 の方法を用いる場合、処理能力 w は

$$w = \frac{kc}{l} \tag{1}$$

で与えられる。 k は CPU の種類、 c はクロック数、 l は Active Process の数に 1 を足したものである。

メモリやキャッシュが十分に大きい場合、その大きさは処理能力に影響を与えない。本稿では多くのメモリを要求するプロセスを走らせることはなく、使用するワークステーションには十分大きなメモリがあるため、 (1) 式では、メモリの大きさは処理速度に影響を及ぼさないことになっている。

負荷分散を行なうためには、コンピュータの処理能力とジョブの重さの 2 つが必要である。処理能力は (1) 式で与えられる。ジョブの重さを決めるにあたって、効率よく負荷分散を行なうためには、次に述べる条件を満たさなければならない。

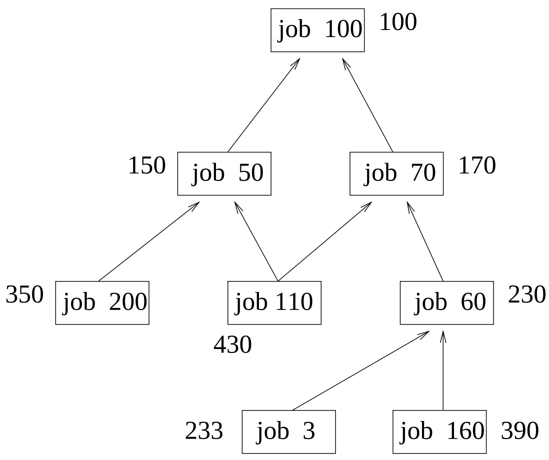

図 4: ジョブの重さ

1. ジョブを DAG から切り離し、それだけで見た時、終了までにかかる時間とジョブの重さは比例する
2. そのジョブが終った後に処理するジョブが重いほど、ジョブの重さは大きい

1 番めの条件は当然であるが、 2 番めの条件は、ジョブが終った後に処理するジョブが重いほど優先的に処理したい、という要求に答えるものである。図 4では、ジョブの重さは自分に依存しているジョブの重さの総和に自分の重さを加えたもの、としている。こうすれば、 2 番めの条件を満たすことができる。

コンピュータの処理能力とジョブの重さがわかれば、後は均等にサーバにジョブを分配すればよい。しかし、この問題は NP-complete [8] であり、正確には解けないため、本稿では次のように行なう。あるジョブがあるマシンに投げられるとすると、その処理能力の逆比とジョブの重さの積を、そのマシンに投げられた最終的なジョブの重さとする。図 5のように、マシン毎にこれらの和をとりつつ、重いジョブから順に、その和が最も小さいものに投げる。

4 評価

4、 8、 16、 32 台のワークステーション (SS20) を用いて、ランダム、 piggyback、 SSS-Server の 3 種類の負荷分散の方法での実行時間を測定した (表 1)。ただし、これらの数値は 5 回の実行時間の算術平均値である。

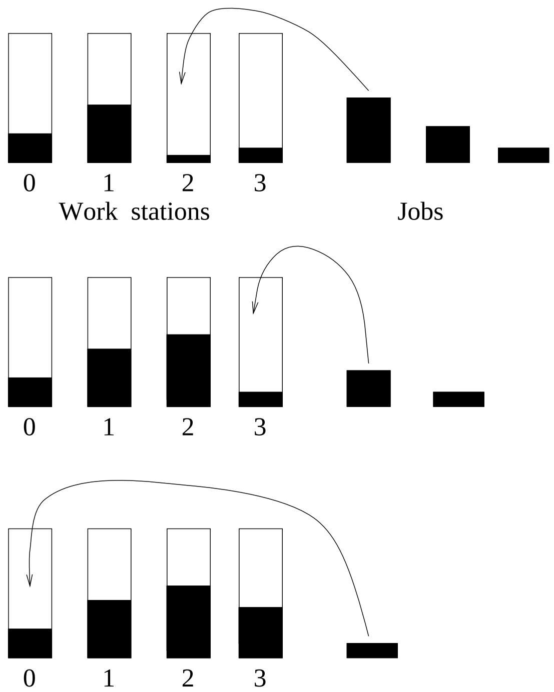

図 5: ジョブの分配

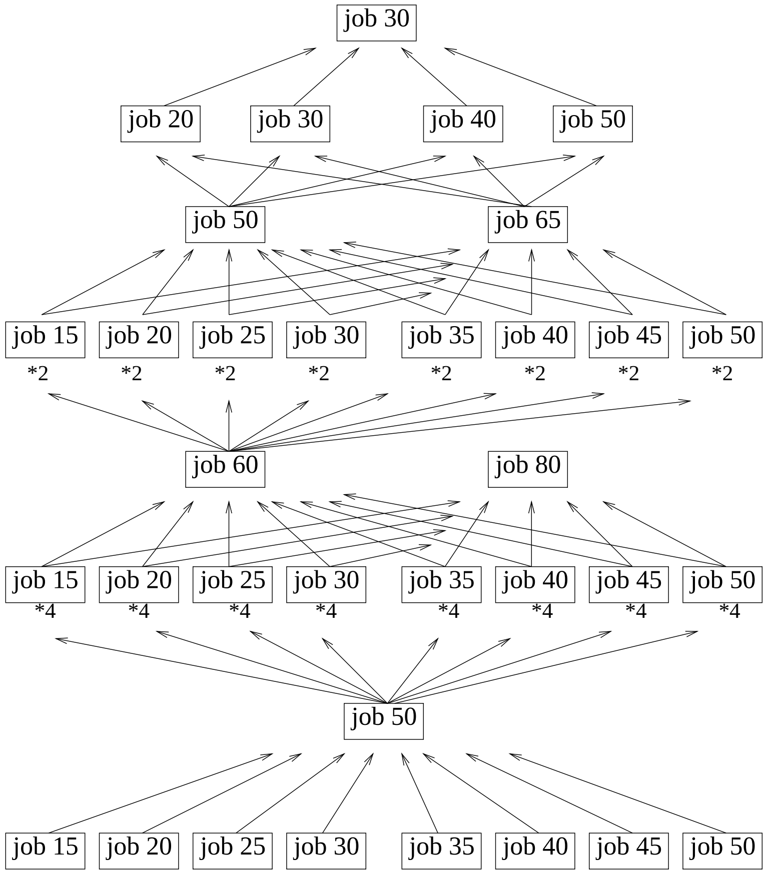

図 6: 実行したジョブ

4.1 評価方法

実行したジョブは、図 6に示した。*2,*4 というのは、そのジョブが 2 個、4 個あることを表す。このようなジョブにした理由は、ワークステーションの数によってジョブの密度を変えたいためと、並列度がそれほど高くないことが実際のジョブではよくあるからである。

また、動的にワークステーションの負荷を変えるにあたって、4 台の時は、job 20,job 25,job 30,job 40,job 50 を、8,16,32 台の時は、このジョブの 2 倍、4 倍、8 倍のジョブを投げた。どのワークステーションにどのジョブが投げられるかを決定するには乱数を用いた。ただし、1:2:3:4 の比で各ワークステーションには重みがつけられ、どの重さになるかはジョブの実行毎に変わる。

4.2 考察

表 1を見ると、4 台のワークステーションでの実行では、SSS-Server が若干速い程度で、あまり差がない。これは、ワークステーションの数に対してジョブが多過ぎるためである (本稿の実験では、最大で、平均 8 個のジョブが 1 台に投げられる)。このような状態では、ワークステーション間の負荷の差が、Active なプロセスの数にして 1、2 個である場合、情報取得とスケジューリングのオーバヘッドが、ランダムにジョブを投げる、結局はほぼ均等にジョブを投げる、ことによる効率的でないスケジューリングのオーバヘッドとつりあってしまう。また、ジョブの実行時間が長いため、ジョブを投げた後に負荷が変化することが多いことも、差がない理由である。

8 台での実行では、ランダムにジョブを投げると非常に遅くなる。このとき、最大で、平均 4 個のジョブが投げられるが、ジョブの数がこの程度だと、外部からの動的な負荷の変化が効いてくる。また、実行時間が短くなったにも関わらず、4 台のとき以上に SSS-Server と piggyback の差があるのは、ジョブの密度が薄くなったために負荷情報の更新があまり行なわれなくなったためである。SSS-Server を走らせているオーバヘッドよりも、不正確な情報によるスケジューリングミスが大きかったことになる。台数による実行速度の短縮がそれほど大きくないのは、図 6の依存関係の形のためである。

16、32 台での実行では、piggyback が遅い。これは、ジョブの密度がさらに薄くなったために、負荷情報がより不正確になったためである。さらに、最大で、平均 1、2 個のジョブが投げられる程度であれば、負荷分散を行なうことによるオーバヘッドは、ランダムに投げることによるスケジューリングミスと大

表 1: 結果

	4	8	16	32
random	142.73	124.52	89.85	85.67
piggyback	142.59	106.17	105.67	96.82
SSS-Server	141.17	104.17	88.49	84.96

差ないことが、SSS-Server と random を比べると分かる。

表 1からは分からないが、ランダムにジョブを投げると実行時間のばらつきが大きかった。また、piggyback を行なうと、情報の正確さが通信量に依存してしまう。全体を通して SSS-Server の実行時間は最も短く、これは、たとえ SSS-Server にとって不利な状況であっても、ワークステーションクラスタ上で負荷分散を行なうならば、SSS-Server が有効であることを示す。

5 おわりに

本稿では、ワークステーションクラスタ上で、依存関係のあるジョブの負荷分散を行なうサーバ、SDA を実装し、負荷分散における SSS-Server の有効性を示した。評価の結果、SSS-Server はワークステーションクラスタ上での動的な負荷分散を効果的に支援することが分かった。これより、SDA と SSS-Server を組み合わせれば、効率良くワークステーションクラスタ透過なジョブの実行が可能であることも分かった。今後の課題としては、以下のことが挙げられる。

- ワークステーションクラスタ透過なジョブの実行のために、SDA にジョブを渡すユーザインターフェイスの実装
- 異なる状況下でのデータの採取と、その評価

6 謝辞

本研究は情報処理振興事業会 (IPA) が実施している独創的情報技術育生事業の一貫として行なった。

参考文献

[1] 亀沢 寛之: 資源情報流通サーバ SSS-Server. 東京大学学士論文, (1996).

[2] 松本 尚、古荘 進一、平木 敬: 汎用超並列オペレーティングシステム SSS–CORE の資源管理方式. 日本ソフトウェア科学会第 11 回大会論文集, pp.13–16 (May 1994).

[3] 松本 尚、平木 敬: Memory-Based-Processor による分散共有メモリ. 並列処理シンポジウム JSPP'93 論文集,pp.245–252,(1993).

[4] 松本 尚: マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察. 計算機アーキテクチャ研究会報告 No.79-1, 情報処理学会,pp.1-8,(1989).

[5] 富田 眞治: 並列コンピュータ工学. 昭晃堂 (1996).

[6] Leslie Lamport: Time,Clocks,and the Ordering of Events in a Distributed Systems. Communications of the ACM, Vol.21, pp.558–564(1978).

[7] Pankaj Mehra, Benjamin W Wah: LOAD BALANCING. World Scientific(1995).

[8] Hesham Ei-Rewini,Theodore G.Lewis,Hesham H.Ali: TASK SCHEDULING in PARALLEL and DISTRIBUTED SYSTEMS. Prentice Hall(1994).