修　　士　　論　　文

ヘテロな分散環境用makeの実装と大規模自然言語処理ワークフローの実行

Implementation of Make for Heterogeneous Distributed Environments and Execution of Large-scale Workflow of Natural Language Processing

指導教員　**田浦 健次朗 准教授**

東京大学情報理工学系研究科
電子情報学専攻

氏 名　**48-66425 関谷 岳史**

提 出 日　**平成 20 年 2 月 4 日**

概 要

大規模自然言語処理の分野では，モジュールの出力が別のモジュールの入力になっているような，多数のモジュール同士を組み合わせた大規模なワークフローを実行する必要がある．いままでは，そのようなワークフローの記述・実行には GNUmake が用いられてきた．しかし，入力の大規模化，各モジュールでの処理の高度化に伴い，単一のマシンでの実行では処理時間が大きくなってしまっている．そこで，本研究では，make をヘテロな分散環境に分散化して実行するようなシステムを設計・実装した．ワークフローを記述するための言語として，DAGMAN などの記法が存在するが，make には (1) 多くのプログラマがなれている (2) 自然で，多くのプログラマが理解している耐故障モデルがある (3) 外部コマンドの実行に多くを頼る workflow を簡便に記述できる，などの利点がある．

一方，分散化と言っても，ヘテロな環境には様々な既存の技術では対応できない問題がある．特に，ファイアウォールやプライベートアドレスしかもたないようなマシンの存在，利用形態のヘテロ性，スケーラブルなグリッド用ファイル共有システムが存在しないこと，などが挙げられる．make を分散化する既存の技術として，dmake があるが，dmake は LAN で接続された複数マシンでの利用を想定しており，各マシンのファイルシステムが NFS などで共有されている必要がある．また，rsh で直接接続できるようなマシンか，SGE がインストールされており qrsh でコマンドを実行できるようなマシンでのみ利用が可能であり，利用形態が大きく制限されている．

そこで，本システムでは，分散プロセスマネージャにグリッド用ツールの GXP を用いることで，GXP が獲得できるような資源すべてにタスクを分散化できるようにした．GXP では，rsh/ssh のほか，Torque などのバッチキュー経由でも資源を獲得可能である．さらに，ヘテロな分散環境では，各マシンでファイルの共有ができないため，各タスクの入出力ファイルを転送する必要がある．そこで，本システムではファイアウォールや NAT を利用したマシンに対応して，複数ノードを中継してパイプライン状に転送を行うファイル転送システムを設計・実装した．

実験では，実際のヘテロな分散環境である，複数管理サイトにまたがる 9 クラスタを用いて，自然言語処理を用いた Web サービスである MEDIE/Info-Pubmed のためのデータベースの構築を行った．本システムが正しく動作することを確認し，ワークフローの中でも特に計算量が大きい HPSG 構文解析のタスクについて，並列化効率は 64%であることを示した．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　複数クラスタ環境の利用に関する現状と大規模自然言語処理のニーズ

PC クラスタが普及してきたことで，一人のユーザが複数のサイトのアカウントを持ち，利用できることが増えてきた．また，ネットワークが高速化してきたために，それらを協調動作させ，ひとつ大規模な計算資源として利用できる可能性も十分にある．しかし，実際はそれぞれのクラスタを独立したクラスタとして利用しているに留まっているのが現状である．それにはさまざまな理由があると考えられる．それには以下のような理由が考えられる．

- サイトごとに管理ポリシーが異なる．近年，企業などから P2P ファイル共有システムなどで情報流出が問題となり，特にネットワークのセキュリティが厳しくなっている．このため，大学，企業のネットワークの入り口にファイアウォールが設置されていることも多い．通常，クラスタであっても，一般と同じ通信網を使うので，例え研究目的に利用するものであっても管理ポリシーによっては制限をはずしてもらうことができない．また，利用形態についても，rsh/ssh などで直接クラスタの各ノードにログインしてインタラクティブに利用できる環境もあれば，バッチキュー経由でしか利用できないような環境もある．

- 利用できるソフトウェアが制限されている．既存の多くのミドルウェアは，管理者権限でインストールして利用するものがほとんどである．そのため，利用可能なクラスタがあったとしても，利用したいミドルウェアがインストールされていないと，協調利用することができない．

- ファイアウォールなどによるネットワークの制限がある．先に示した理由により，セキュリティの観点から，外部からの通信は制限されているのが普通である．また，クラスタに割り当てられるグローバル IP アドレスにも限りがあるので，ログインのためのノードのみにグローバル IP アドレスが割り振られており，残りのノードはプライベート IP アドレスのみが割り振られている場合もある．クラスタは通常同一 LAN 内に設置され，ファイアウォールなどの制限なしで通信できる．そのため，既存のミドルウェアは各ノードが自由に通信できるという前提で設計されているものが多い．また，WAN をまたいだ通信では，LAN 内での通信に比べ，通信遅延が大きく，通信帯域が小さいのが普通であり，単一クラスタ内での利用を想定して設計されたミドルウェアは大きく性能低下してしまう．

- 複数サイトにまたがると NFS のようなファイルシステムが利用できず，ファイルの共有がめんどうである．既存のクラスタでよく利用されているキューイングシステムや MPI などのシステムは NFS などで各ノードのファイルが共有されているのが前提となっている．WAN をまたいで利用可能である sshfs [13] や gfarm [20] などのファイルシステムも存在するが，sshfs は主にサーバー 1 対クライアント数ノード程度の利用を想定しており，スケーラビリティにかける．gfarm はサーバをファイアウォールの中に置くことはできず，閉じた環境でのみしか利用できない．

一方，自然言語処理の分野では，現在実現されている処理よりも，より高度な言語理解のための研究がすすめられている．例えば，現在の検索は形態素解析をベースとした単語による検索が主流であるが，精度の高い構文解析による関係概念の検索などの技術も研究されている．しかし，精度の高い構文解析は，一般に計算量が莫大であり PC クラスタなどの利用による並列化が必須になってきている．また，近年では Web の発達により，Web 上の文書を対象とした研究も進められており，処理する入力の量も非常に大きくなってきているという側面もある．

東京大学辻井研究室で開発されている，MEDIE/Info-Pubmed [22] [6] [4] は医学文献データベース MEDLINE 用の検索システムで，遺伝子と疾患の相関やたんぱく質の相互作用といった関係概念を検索する Web サービスである．このサービスのために，オフライン処理として MEDLINE の全文献に対して，述語項構造と辞書・オントロジ ID が付与された意味構造付きテキストのデータベースの構築を行う．構築には入力文章に対して，何段階もの自然言語に関する処理 (構文解析，固有名識別など) を行う必要がある．各処理はモジュール化されており，あるモジュールの出力が別のモジュールの入力になっている．全体として，それらが複雑に組み合わさったワークフローにより最終的な出力を生成する．これまではワークフローは基本的に GNU make で処理を行い，特に計算量の大きな部分 (enju[2] による HPSG (Head-Driven Phrase Structure Grammar) 構文解析) のみ取り出し，グリッド用シェル GXP [16] [3] による並列コマンド投入と SCP などのファイル転送を用いて，複数の PC クラスタで並列化を行っていた．しかし，他のモジュールについても並列化すれば処理時間を短縮することが可能であるうえ，ワークフローが一貫して処理できるシステムがあれば，PC クラスタへのデータのコピーなど，人手による操作の負担を小さくすることができる．

1.2 本研究の目的

本論文では前節で述べたようなヘテロな分散環境において，Makefile によって記述されたワークフローを処理するシステムの設計について述べる．また，このシステムの実装に必要な基盤技術と実装について述べる．さらに，このシステムを用いて MEDIE/Info-Pubmed のためのデータベースの構築を行い，評価実験とした．

Make 実行システムは GXP の提供する mw フレームワークという仕組みの上に実装されており，

ノードは GXP で獲得できさえすれば，タスクを処理する構築サーバとして利用可能である．GXP でノードを獲得できるとは，計算ノード上に GXP のプロセスを立ち上げられるということである．GXP では ssh/rsh のほか，torque や SGE などのバッチキュー経由でもノードを獲得可能であるため，様々な運用ポリシーで管理された資源が利用可能である．また，入出力ファイルをノード間でやりとりするために，GXP を用いたデータ転送ツールを実装し，利用している．このツールは，socket での通信のほか，ssh での通信にも対応しており，scp でのみしかデータがコピーできないような，ネットワークの制限が大きい環境でも利用可能である．また，複数のノードをリスト状に繋げて転送も可能であるので，(外部) → (外部からログイン可能なノード) → (プライベートアドレスしか持たないようなノード) のような経路で外部からプライベートアドレスしかもたないようなノードへの，直接のファイル転送も可能である．これらのシステムは，通信の起点となるノード以外で利用する分のプログラムについてはすべて python で記述されている．起点となるノード以外には GXP がノードを獲得した時点でスクリプトファイルがコピーされ，自動的にインストールが行われる．さらには，前提とするソフトは python と計算ノードにログインするための仕組み (ssh/queue) のみであり，管理者による特別なソフトのインストール・設定は必要ない．

大規模自然言語処理での利用を想定すると，計算の実行は長時間になると考えられるため，計算実行中に資源の状態 (予期しない故障・計画的なメンテナンスなどによる停止・それらからの復旧など) が変化する可能性がある．そのような環境で資源を効率的に利用するためには，故障・資源の増減に対応する必要がある．そこで，まずシステムの基盤である GXP に手を加え，動的にプロセスを追加できるように変更したので，これについても本論文で示す．

1.3 本論文の構成

第 2 章 関連研究 分散環境でワークフローを処理するようなシステムと，動的な資源の増減に対応したシステムの先行研究を紹介する．

第 3 章 ヘテロな分散環境で利用するためのシステムに対する要請 ネットワークや利用形態が様々なサイトが集まったようなヘテロな環境では，単一のクラスタ環境に比べ様々な使いづらい要因がある．そのような環境で利用するために必要なことについて述べる．

第 5 章 システムの実装 システム全体の構成や，その構成要素それぞれについての実際の実装の方法を述べる．

第 6 章 実験と評価 本システムを実環境で実験した結果と，その評価について述べる．

第 7 章 おわりに 本論文のまとめと，今後の課題について述べる．

第2章　関連研究

2.1　動的な資源の増減に対応したシステム

本論文での動的な資源の参加・脱退とは，複数プロセスが協調して計算を開始した後に，プロセスが増減するようなことを指す．並列計算のための通信モデルにはいくつかの種類があるが，それぞれに参加・脱退の難しさが異なる．

マスター・ワーカーモデルでは，1 つのマスタープロセスと複数のワーカープロセスからなるモデルであり，ワーカーはマスターとのみ通信を行う．そのため，マスターがすべてのワーカーの状態を完全に把握可能であり，簡単にワーカーの参加脱退を行うことができる．また，通信プロトコルも単純にできるという利点がある．ただし，欠点としてはマスターは脱退することはできない，マスターが通信のボトルネックになる，などが挙げられる．マスター・ワーカーモデルを拡張して階層型にしたものに Jojo2[21] がある．Jojo2 はマスター・サブマスター・ワーカーの 3 種類のプロセスからなり，マスターはサブマスターと，サブマスターはワーカーと通信を行う．これは，サブマスターを各クラスタのある 1 ノード (ログインノード)・ワーカーを計算ノードに配置することで，複数クラスタでの利用を想定している．グリッド環境で通信に通常のマスターワーカーモデルを用いると，マスタープロセスとワーカープロセスがネットワーク的に遠い場所に配置されてしまうという欠点があるが，階層的なモデルにすることで，サブマスター・ワーカー間の通信については，同一 LAN 内の通信になり，性能が向上する可能性がある．また，通常のマスター・ワーカーモデルでは，マスターにすべてのワーカーが接続することになるので，スケーラビリティに欠ける．一方，階層的にすることで，本質的には同じことではあるが，現状のクラスタの規模であれば十分スケールする．

メッセージパッシングモデルでは，任意の参加プロセス同士が，通信を行う．代表的な規格には MPI があり，MPICH[7] などの実装がある．MPI では各プロセスにはランクと呼ばれる ID が割り振られ，その ID に向けてメッセージを送ることで，特定の相手との通信を行うことができるというモデルになっている．各プロセスが対等の関係で，マスターなどのボトルネックや Single Point of Failuer になりうるプロセスがいないため，スケーラビリティが高いといえる．一方，プロセスの参加・脱退という観点から見ると，ID の数 (プロセスの数) は静的に決まるため，ID が計算中に増えたり減ったりすることは不可能である．そこで，Phoenix[17] では，ID の代わりに十分大きな仮想プロセッサ名空間というものを用意し，実際のプロセスにそれらを割り当てる．参加・脱退をする際にはそれらをやり取りすることで，ID に向けて送ったメッセージの配送を破綻させることなく，メッ

セージパッシングモデルでのプログラミングを可能にしている．Phoenix では，接続ツリーのリーフのノードのみしか脱退することはできなかったが，我々の以前の研究 [19] では，任意のプロセスが脱退するためのプロトコルを提案している．しかし，Phoenix モデルでのプログラミングは難しく，まだ実用的なアプリケーションで利用された実績がない．

本研究では GXP の通信に関しては，簡単な脱退を可能にするため，階層的な接続 (ルートノード→クラスタのヘッドノード→クラスタの計算ノード) で利用することを想定している．また，システムは GXP の与えるマスターワーカーフレームワークである mw フレームワーク (実際の通信は単純なマスター・ワーカーでなくてもよい (4.2.4 節参照)) を利用する．

2.2 グリッド環境のためのツール

本システムではグリッド環境のヘテロ性を隠匿し，透過的に利用するために GXP を利用している．GXP については 4 節で詳しく述べる．

Condor[12] は複数のクラスタや PC を利用するためのスケジューリングシステムである．Condor は元々遊休 PC を利用するために開発されたシステムで，GXP と同様に，WAN をまたいだ様々な利用形態のクラスタを利用することが可能であり，例えば PBS などのクラスタのスケジューラや Globas などを通して，資源を利用することができる．また，これらの資源は動的に参加・脱退可能である．一方，大きく異なるのは，Condor 自体が複数ユーザを調整するスケジューラであるということである．構成としては，サブミットマシン・セントラルマネージャ・実行マシンがある．サブミットマシンから投入されたタスクを，あらかじめ設定された実行マシンのプールからセントラルマネージャが適切なものを選び，割り当てる．このためには，セントラルマネージャはもちろん，各実行ノードにも管理者権限でのインストール・適切なネットワークの設定が必要であり，ユーザ権限でのログインができればよい GXP に比べ，資源の選択の幅は縮まる．また，一般に単一クラスタでのスケジューリングに比べ，グリッドでのユーザ間のフェアなスケジューリングは難しい．一方 GXP は特定のユーザアカウントで動くため，各クラスタで利己的に資源を利用してよく，グリッド全体を見た調整などは必要ない (最も，各クラスタの利用ルールは守る必要があるが，これは動的なプロセスの参加・脱退がサポートされていれば，比較的簡単にルールを守る動作をすることが可能であると考えられる)．現在サポートはされていないが，理論的には Condor の上で GXP を立ち上げて，(Condor プール)+(Condor のインストールされていないクラスタ) のような利用も可能である．

2.3 ワークフローを分散して処理するシステム

ワークフローを分散して実行する既存のシステムとしては次のようなものがあげられる．

DAGMAN[1] は Condor に付随したワークフローエンジンで，フローグラフが DAG になるようなワークフローを記述可能である．各ジョブは 1 ファイルに 1 ジョブという形でユーザが定義し，ジョ

ブ間の依存関係を dag file に記述する．DAGMAN は dag file を解析し，依存する JOB の実行がすべて完了した JOB を Condor Scheduler に渡す．ジョブが失敗した場合には，あらかじめ指定した回数再試行を行う．ただし，dag file は静的に解析されるため，動的にタスクが生成されるような場合には対応ができない．さらに，condor pool のノードの故障には対応しているものの，DAGMAN daemon 自身が故障してしまった場合には，すべての途中結果が破棄されてしまうという問題もあると考えられる．より抽象的な (割り当てられるある特定のリソースを指定していない) ワークフローを，DAGMAN に渡す Pegasus [10] というツールもある．

Taverna[15] は複数の WEB サービスをつなげてワークフローを構築することができる GUI ベースのツールである．主にバイオインフォマティクスなどの分野で利用されている．GUI であるため直感的にワークフローを記述できるという利点がある．しかし，管理ドメインのポリシー・ネットワークの状況によっては，すべての計算ノードで WEB サービスを提供することができるとは限らず，ヘテロな分散環境での利用は難しい．

DMAKE[8] は make を分散化したもので，大規模なプロジェクトの構築を複数のサーバ (構築サーバ) に分散して行うものである．Makefile を解析して，並列して実行できるターゲットを判別する．各ノードのファイルシステムは NFS などで共有されている必要があり単一クラスタのみでしか利用ができない．また，rsh 経由で使う場合には，各ノードには */etc/opt/SPROdmake/dmake.conf* という設定ファイルが必要であり，管理者権限での設定が必要である．rsh 使わない場合にはクラスタに SGE がインストールされている必要がある．今回の実装では，一部に dmake を用いている．

GridAnt [9] は，Java 用のビルドツールである Ant の記法を用いてワークフローを記述し，処理するシステムである．Java CoG Kit [14] と呼ばれるグリッド環境とのインターフェースを通して，Globus Toolkit [11] によって，グリッド資源を利用する．またファイル転送は GridFTP のプロトコルを用いて，計算ノード間での転送を行う．

本システムはワークフローを記述するための言語として make を利用している．make には (1) 多くのプログラマがなれている (2) 自然で，多くのプログラマが理解している耐故障モデルがある (3) 外部コマンドの実行に多くを頼る workflow を簡便に記述できる，などの利点がある．make を分散化するシステムには DMAKE がある．DMAKE は SGE か rsh のみでしかタスクを生成できないが，本システムでは GXP で獲得できるようなノードすべてを利用できる．また，DMAKE はすべての構築サーバで，NFS などによって共有されたディレクトリが必要であるが，本システムではファイル転送を行うため必要ない．さらに，管理者権限を必要とするインストール・設定が必要ないという利点もある．

第3章　ヘテロな分散環境で利用するためのシステムに対する要請

1.1 で述べたような，ヘテロな分散環境で利用するためのシステムには様々な要請がある．

3.1　スケーラビリティ

複数クラスタを利用するような大規模なシステムはスケーラブルな構成，機能を備える必要がある．例えば，ルートノードから直接すべての構築サーバへ接続を張るようなシステムでは，規模が大きくなるにつれ，ルートノードに通信の負荷が集中し，性能のボトルネックになりやすい．

3.2　資源の増減・故障への対応

単一時間内にシステムの構成要素 (マシン・ネットワーク)i が故障する確率を $P_f(i)$，構成要素の数を n，実行時間を t として，システムの構成要素がひとつでも故障する確率 (システムが故障に遭遇する確率) は

$$P_{fTotal} = t(1 - \prod_{i=1}^{n}(1 - P_f(i)))$$

と表すことができる．$P_f(i)$ が t に対して一定であると仮定すると，システムが故障に遭遇する確率は，資源の規模と，実行時間に対して単調増加する．そのため，資源の規模も，実行時間も大きくなるような大規模長時間計算では故障に対応することが重要である．ただ，故障への対応といっても，様々なレベルでの対応がある．これは対象とするアプリケーションによるが，

1. 故障が起きるとそれを受動的に検知するシステム．プロセス間は普通，socket を利用して通信を行うため，OS が通知するエラー (例えば，TCP ソケットであれば connection reset by peer や broken pipe など) を検出する．ただし，send() や recv() を呼び出すまでは，このようなエラーは通知されないので，それまで故障は分からない．

2. 故障を能動的に検知するシステム．これは例えば，堀田らの手法 [23] が挙げられる．この手法では，各ノードがランダムに選んだいくつかの他のノードへ監視を依頼し，定期的にハートビートを送る．そして，もし，ある一定期間ハートビートが届かなかった場合には，故障とみ

なすという手法である．また，この手法では監視のために構築したグラフを利用し，故障の情報を高速に全体に伝播している．

3. 状態のチェックポイントをとり，故障の直前から再実行するシステム．タスクやメモリやバーチャルマシンなどチェックポイントを取るレベルも様々である．後者になるほど汎用性が高まるが，チェックポイントのサイズが大きくなる．また，いつ，チェックポイントを取るかや，どのポイントまで巻き戻してよいか，などの判断は並列計算では一般に難しく，一部のシステムで利用されているに留まる．

また，複数のクラスタを利用する場合には，すべてのクラスタが万全の体制で利用できるということは少なく，例えば，メンテナンスによる計画的な停止や，それらからの復旧などが実行中に起こりうる．また，クラスタは複数ユーザによって共用されている場合がほとんどで，資源を一人のユーザで占拠してしまわないよう，システム的または紳士協定によって利用時間の制限が設けられている場合もある．このような資源を効率的に利用するためには，一定時間利用したあとに，その資源からはプロセスを脱退させるような仕組みや，途中からプロセスを参加させる仕組みが必要になる．

3.3 環境のヘテロ性への対応

クラスタの各管理ドメインのセキュリティポリシーやネットワーク事情は様々である．クラスタによってはNATを利用しているためにクラスタ外からの通信が行えなかったり，セキュリティを高めるためにファイアウォールによって，ssh などの最小限のポート以外への通信を遮断していたりする可能性がある．また，インストールされているソフトウェアがクラスタ間で異なっていたり，計算ノードの利用方法 (バッチキュー経由・rsh/ssh 経由など) が異なる場合もある．そのような環境でのプロセスマネジメント・ファイル転送の仕組みが必要であり，しかも，それらが管理者権限を必要とせずユーザレベルでインストールが行えるとより利便性が高い．

3.4 入出力ファイルのマネジメント

単一クラスタ環境ではNFSなどのネットワーク共有ファイルシステムにより，各マシンが同じファイルにアクセスすることができ，ソフトウェア環境の共通化や，入出力ファイルの共有が行える．上記のようなヘテロなグリッド環境では，今のところユーザレベルで利用できるそのようなシステムはないため，何らかの形で入出力ファイルを転送する必要がある．

第4章　GXPの動的な資源の増減への対応

4.1　GXPの構成

GXP とは，グリッド用ツールであり，複数ノードにプロセスを一度に立ち上げたり，それらのプロセス間で簡単に通信を行える仕組みを提供している．構成は図 4.1 のようになっている．

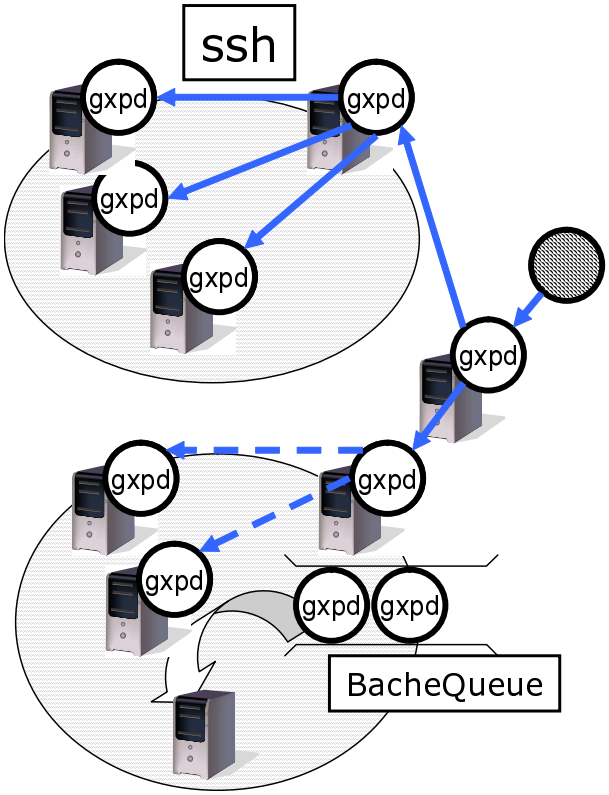

図 4.1: GXP の構成

gxpc が GXP に命令を与えるプロセスで，ユーザがコマンドを実行するたびに新しく生成されるプロセスである．gxpd が，GXP が獲得したノードに立ち上がるデーモンプロセスで，gxpd 同士はツリー状に接続が行われる．つまり GXP はツリー状にノードを獲得していく．gxpd は図のように

sshd を通して立ち上げられたり，Torque などのバッチキューで立ち上げることも可能である．

4.2 GXP の利用

ここでは GXP の基本的な利用法について紹介する．

まず，プロンプトに GXP の情報を表示するため，

```
export PS1='[\u@\h:\w]`gxpc prompt 2> /dev/null`% '
```

のように環境変数を設定する (bash の場合)．

以下は，hongo100 というノードで上記のように環境変数を指定した sekiya ユーザが gxp を立ち上げたときのログである．

```
1 [sekiya@hongo100:~]% gxpc
2 gxpc: no daemon found, create one
3 /tmp/gxp-sekiya-default/gxpsession-hongo100-sekiya-2008-02-02-12-02-44-9018-164\
4 75558
5 [sekiya@hongo100:~][1/1/1]%
```

4 行目のプロンプトに表示されている 3 つの数字は，左から順に，直前のコマンドが成功したノード集合 P_{ok} の数，GXP がコマンドを実行するために選択しているノード集合 P_{exec} の数，GXP が獲得しているノード集合 P_{peer} の数，を示している．

$$P_{ok} \subset P_{exec} \subset P_{peer}$$

の関係が成り立つ．

4.2.1 use: ノードの獲得方法の指定

use コマンドは GXP にノードの獲得方法を通知するためのコマンドである．

```
1 [sekiya@hongo100:~][1/1/1]% gxpc use ssh hongo hongo
```

とすると，ホスト名の先頭が hongo にマッチするノードからホスト名の先頭が hongo にマッチするノードへのノード獲得には，ssh を用いるという指定である．

```
1 [sekiya@hongo100:~][1/1/1]% gxpc use torque hongo hongo-torque
```

とすると，hongo-torque というノードの獲得には hongo から torque 経由で行うという指定である．

4.2.2 explore: ノードの獲得

explore コマンドは use で指定したログイン方法で実際にノードの獲得を行うコマンドである．

```
1  [sekiya@hongo100:~][1/1/1]% gxpc explore hongo101
2  reached : hongo101
3  [sekiya@hongo100:~][2/2/2]% gxpc explore hongo[[102-110]]
4  reached : hongo102
5  reached : hongo105
6  reached : hongo104
7  reached : hongo103
8  reached : hongo107
9  reached : hongo106
10 reached : hongo109
11 reached : hongo108
12 reached : hongo110
13 [sekiya@hongo100:~][11/11/11]% explore hongo-torque
14 reached : hongo-torque
15 [sekiya@hongo100:~][12/12/12]% explore hongo-torque 10
16 reached : hongo-torque
17 reached : hongo-torque
18 reached : hongo-torque
19 reached : hongo-torque
20 reached : hongo-torque
21 reached : hongo-torque
22 reached : hongo-torque
23 reached : hongo-torque
24 reached : hongo-torque
25 [sekiya@hongo100:~][21/21/21]%
```

1 行目では hongo101 番の獲得を試みている．use コマンドで hongo から hongo へは ssh でノードを獲得するという指定をしているので，ssh で hongo101 を獲得する．2 行目で獲得できたことが示されている．3 行目のプロンプトには，[2/2/2] という風に表示され，獲得されているノード数が 2 であることを示している．さらに，3 行目のように数字を [[]] で囲むと，連続した数字に展開され，複数の数字が連続したノードを一度に獲得することができる．13 行目では同様に torque 経由でノードを獲得している．15 行目のように後ろに数字をつけると，同じノードに複数回のログインを試みる．ただし，torque 経由の場合，同じノードを指定しても，torque の割り当てによるので，別々のノードになる．(同じノードであることもある．)

4.2.3 e: コマンドの実行

e コマンドは選択されたノード P_{exec} で任意のコマンドを実行するためのコマンドである．

```
1  [sekiya@hongo100:~][21/21/21]% gxpc e hostname
2  hongo100
3  hongo102
4  hongo107
5  hongo106
6  hongo105
7  hongo103
8  hongo101
9  hongo108
10 hongo110
```

```
11 hongo109
12 hongo007
13 hongo010
14 hongo004
15 hongo003
16 hongo104
17 hongo006
18 hongo009
19 hongo002
20 hongo001
21 hongo011
22 hongo008
23 [sekiya@hongo100:~][21/21/21]% gxpc e 'hostname␣|grep␣hongo0'
24 hongo008
25 hongo007
26 hongo009
27 hongo011
28 hongo002
29 hongo004
30 hongo006
31 hongo010
32 hongo003
33 hongo001
34 [sekiya@hongo100:~][10/21/21]%
```

1 行目のように，e のあとに続けて，実行したいコマンドを指定する．すると，各ノードで指定したコマンドが実行され，それらの出力 (標準出力・標準エラー出力に書き出された出力) がばらばらに出力される．23 行目のプロンプトでは P_{ok} の数が 21，すなわち，すべてのノードでコマンドが成功 (コマンドの返り値が 0) したことを示している．さらに 23 行目のように，ホスト名に hongo0 が含まれるようなノードのみ成功するようなコマンドを実行してみると，34 行目のように P_{ok} の数は 10 になっている．なお，e コマンドで実行したいコマンドに，空白や環境変数や，ワイルドカードなどのシェルで展開されてしまうものを含めるには” ’ ”(single quote) で囲う．

4.2.4 mw: mw フレームワーク

mw コマンドは GXP が与える mw フレームワークを実行するためのコマンドである．

```
1 [sekiya@hongo100:~][10/21/21]% mw −−master ./master.py ./worker.py
```

のように実行する．ひとつの master.py が hongo100 で実行され，P_{exec} のノードで worker.py が実行される．master.py の標準出力 (ファイルディスクリプタ 1 番) への書き込みが，各 worker.py のファイルディスクリプタ 4 番から読み出し可能になる．また，同様に，各 worker.py のファイルディスクリプタ 3 番への書き込みが master.py の標準入力 (ファイルディスクリプタ 0 番) から読み出し可能になる (図 4.2).

図 4.3 は mw フレームワークを用いて，タスクを master からワーカーに振り分けるプログラム (python 風の擬似コード) である．マスターは 1 行標準入力から 1 行読み込み (8 行目)，それが，どのワーカから来たものか，どんなメッセージかをパースする (9 行目)．もし，それがタスクのリクエ

ストだった場合には (10 行目)，タスクを生成し (11 行目)，それをワーカーの名前とともに，string に変換して，改行文字をつけて，標準出力に書き出す (12 行目).

ワーカーは，自分の名前を乗せたタスクのリクエストメッセージを作り (8 行目)，それを改行文字とともに string に変換してファイルディスクリプタ 3 番へ書き出す (9 行目)．次に，ファイルディスクリプタ 4 番から 1 行読み出し (12 行目)，もしそれが，自分宛のメッセージだった場合には (14 行目)，メッセージに乗っていたタスクを実行する (15 行目).

使用する上で，いくつかの注意すべき点がある．まず，通信はファイルディスクリプタを通して行うため，メッセージをすべて，文字列に変換して行う．また，ワーカーからの書き込みは，すべて混じってしまうため，メッセージを 1 行になるようにして利用する．さらに，ファイルディスクリプタは通常バッファされるので，flush() を利用して，バッファから確実に書き出されるようにする．今回は python 風に記述したが，ファイルディスクリプタを読み書きできるものであれば，なんでもよい.

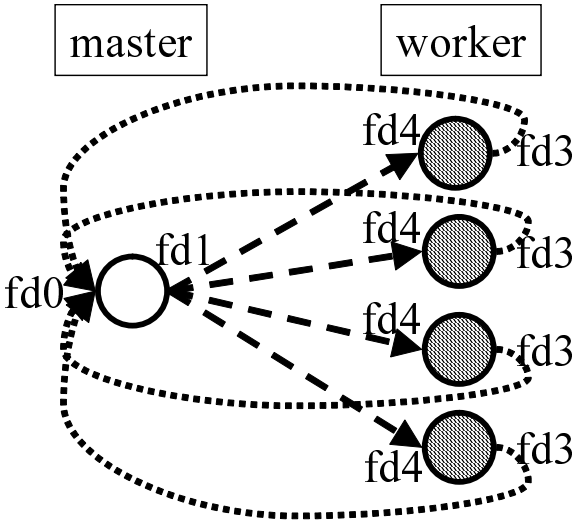

図 4.2: mw フレームワーク

4.2.5 smask: ノードを選択する

smask コマンドは実行するためのノード P_{exec} を選択するコマンドである．例えば，hongo0??というホストのみを選択したい場合には,

```
1 [sekiya@hongo100:~][21/21/21]% gxpc e 'hostname␣|grep␣hongo0'
2 hongo008
3 hongo007
4 hongo009
5 hongo011
6 hongo002
```

```
1  import sys
2
3  to_worker = sys.stdout
4  from_worker = sys.stdin
5
6  def master():
7      while True:
8          line = from_worker.readline()
9          sender, message = parse_line(line)
10         if message is request_of_task:
11             task = make_task()
12             to_worker.write(string_of(sender, task)+'\n')
13             to_worker.flush()
```

```
1  import os
2
3  to_master = os.fdopen(3, 'w')
4  from_master = os.fdopen(3, 'r')
5
6  def worker():
7      while True:
8          message = request_of_task(my_name)
9          to_master.write(string_of(message)+'\n')
10         to_master.flush()
11         while True:
12             line = from_master.readline()
13             message = parse_line(line)
14             if message is to_me:
15                 status = do_task(message)
16                 break
```

図 4.3: mw フレームワークを用いた例

```
 7 hongo004
 8 hongo006
 9 hongo010
10 hongo003
11 hongo001
12 [sekiya@hongo100:~][10/21/21]% gxpc smask
13 [sekiya@hongo100:~][10/10/21]%
```

e コマンドでそのようなホストのみが成功するようなコマンドを実行し，$P_{ok} = p|p$ は *hostname* に *hongo*0 を含むとなるようにする．次に 12 行目のように smaks コマンドを実行する．すると，13 行目のプロンプトにあるように，$P_{exec} = P_{ok}$ となり，以降は 10 ホストでのみ e コマンドなどが実行されるようになる．なお，これを解除する ($P_{exec} = P_{peer}$ にする) には，rmask コマンドを実行する．

4.2.6 trim: 選択されたノード以外のノードを解放する

trim コマンドは，一度獲得したノードを解放するためのコマンドである．

```
 1 [sekiya@hongo100:~][10/10/21]% gxpc trim
 2 hongo100-sekiya-2008-02-02-12-02-44-9018 : child hongo107-sekiya-2008-02-02-12-\
 3 26-40-4403 hongo106-sekiya-2008-02-02-12-26-40-20268 hongo102-sekiya-2008-02-02\
 4 -12-26-39-16076 hongo101-sekiya-2008-02-02-12-26-25-23646 hongo108-sekiya
       -2008-\
 5 02-02-12-26-40-18237 hongo103-sekiya-2008-02-02-12-26-40-29356 hongo109-sekiya
       -\
 6 2008-02-02-12-26-40-28190 hongo105-sekiya-2008-02-02-12-26-39-1597 hongo110-sek
       \
 7 iya-2008-02-02-12-26-40-43555 will be trimmed
 8 [sekiya@hongo100:~][12/12/12]% e hostname
 9 hongo100
10 hongo003
11 hongo009
12 hongo011
13 hongo002
14 hongo007
15 hongo010
16 hongo004
17 hongo001
18 hongo006
19 hongo008
20 hongo104
21 [sekiya@hongo100:~][12/12/12]%
```

1 行目のように smask によって P_{exec} が 10 になっている状態で，trim コマンドを実行する．すると 8 行目のように 12 ノードが獲得された状態になる．trim コマンドは基本的に $P_{peer} = P_{exec}$ となるようにするコマンドであるが，GXP は接続をツリー状に行うため，たとえ P_{exec} に含まれていないノードであっても，子ノードが P_{exec} に含まれている場合，解放してしまうと，子ノードも解放されてしまうことになるため，このようなノードは解放されない．すなわち，P_{peer} は GXP の接続木の部分木の内，すべての $p \in P_{exec}$ が含まれるような最小の部分木になる．

4.3 GXP の拡張

本システムで利用するために，GXP で e コマンドや mw コマンドの実行中にプロセスを追加できるように拡張を行った．

gxp には session という概念があり，これは，現在の獲得しているノードや，選択されているノードなどの情報を含んでいる．gxpc コマンドが終了するたびに，セッションファイル，*/tmp/gxp-USERNAME-default/gxpsession-SUFFIX* (*USERNAME* は実行ユーザのユーザネーム，*SUFFIX* はホスト名や実行日時などを含んだ文字列) というファイルに書き出される．gxpc が一度でも実行されたノードではこのファイルが作成され，以降はこのファイルを読み出してからコマンドが実行される．

e コマンドに関しても同様で，セッションファイルを読み出し，コマンドを実行し，実行結果を同じファイルに書き戻すという仕組みになっている．つまり，セッションファイルを読み出すのは 1 度きりなので，e コマンドの実行中に P_{exec} を変更するには，セッションファイル以外の方法で，変更されたことを e コマンドに伝える必要がある．これを実現するために，e コマンドを実行している gxpc に対して，他の gxpc から gxpd を通して情報を伝えることを行うように変更した．explore コマンドが実行されたときには，P_{peer} が変更されたことを伝える必要があり，すでに実行されている e コマンドにノードを追加したい場合には P_{exec} が更新されたことを伝える必要がある．

explore コマンドでノードを新しく獲得し $P_{peer} = P^*_{peer}$ になった場合には，他の e コマンドに P^*_{peer} を伝える (図 4.4)．e コマンドで新しいノードでコマンドを実行し，それを既存の e コマンドと一緒に扱いたい場合 (e –join 〜) には，P^*_{exec} を伝える (図 4.4)．

e コマンドはノードからの出力を gxpc の標準出力に出しているが，それを別に立ち上げた MASTER プロセスに渡すように変更したものが mw コマンドである．そのため，mw コマンドについても e コマンドと同様に動作しているので，これらの仕組みは mw コマンドでも利用することが可能である．

4.3.1 予備実験

予備実験として，GXP で構文解析の多数のタスクを mw コマンドで実行中に，クラスタを故障させ，それをすぐに復帰させるという状況で GXP が正しく動作するかの実験を行った．実際にはクラスタをとめることはできないので，クラスタのヘッドノードの gxpd プロセスを kill することでクラスタの故障を擬似的に発生させた．クラスタは hongo，chiba，imade という 3 つのクラスタを用いた．

結果は図 4.5 のようになった．グラフは横軸が時間で，縦軸が各クラスタのノードの CPU 使用率の合計である．hongo クラスタが約 18 秒後に故障し，すぐ復帰，chiba クラスタが約 80 秒後に故障し，すぐ復帰する，というストーリーである．故障の後，gxpc ping で故障していないノードを選択し，gxpc trim で死んでしまったノードを切り離す．そのあと，gxpc explore で故障して復帰したク

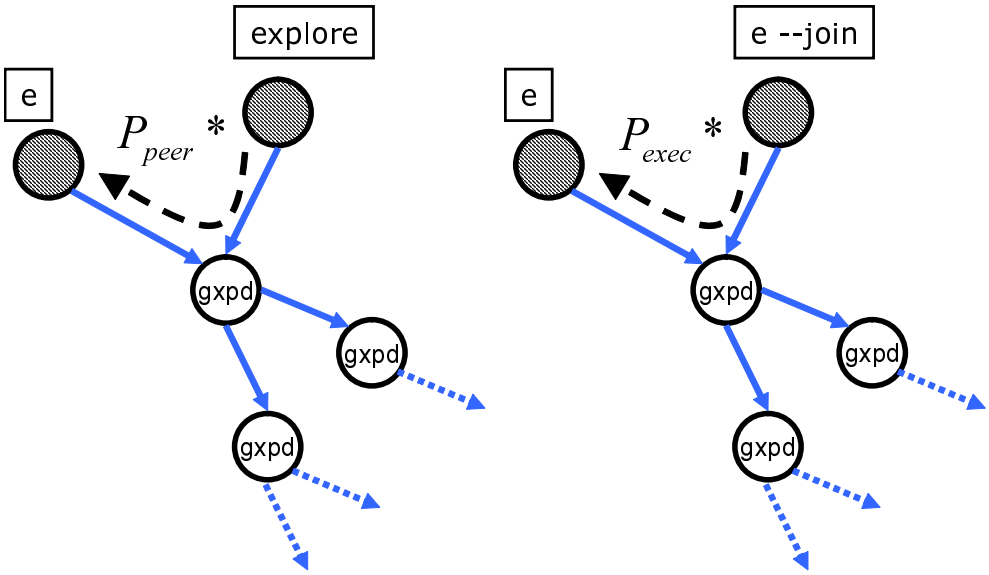

図 4.4: 動的な explore コマンドと e コマンドによる join

ラスタのノードを再び獲得し，gxpc mw –join ～で既存の mw コマンドと通信を行えるようにしながら，新しいノードにプロセスを立ち上げる．故障の後，CPU 使用率が下がっているが，50 秒程度の後，CPU 使用率が回復していることが分かる．もし，既存の mw コマンドに正しく join できなければ，タスクを得ることができず，CPU 使用率が上がらないので，正しく join も行えていることが確認できた．また，50 秒程度 CPU 使用率が回復するのにかかっているのは，ほぼ explore にかかる時間で，torque でノードを獲得するのにかかる時間であった．

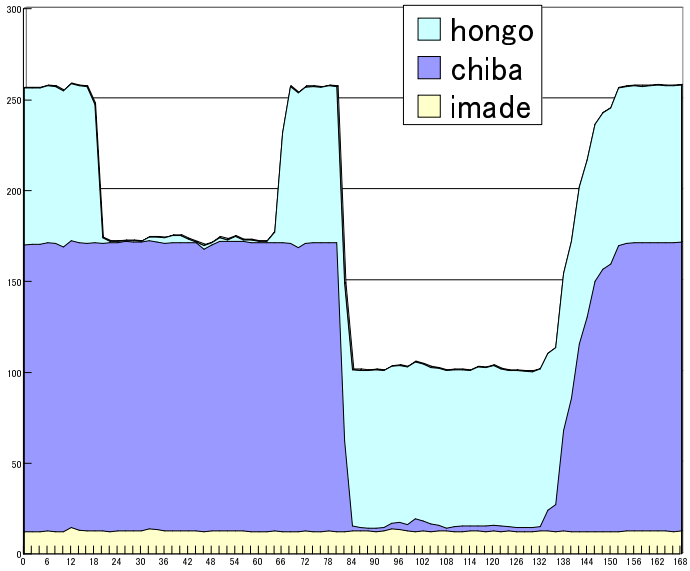
hongo
chiba
imade
300
250
200
150
100
50
0
0 6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96 102 108 114 120 126 132 138 144 150 156 162 168

図 4.5: 参加・故障実験

第5章　Make実行システムの実装

5.1　構成

システムの構成は図 5.1 のようになっている．

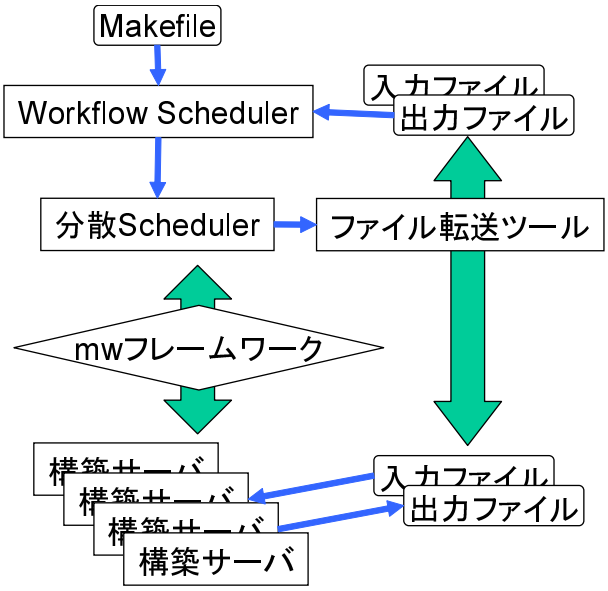

図 5.1: システム構成

最も下のレイヤにはプロセスマネージャとして GXP を用いた．GXP の与える mw フレームワークの上にワーカーとして構築サーバ，マスターとして分散スケジューラを立ち上げる．ワークフロースケジューラは Makefile を解析し，現在実行可能なターゲットを見つけるプロセスで，分散スケジューラは，タスクをどのノードへ割り当てるかを決めるプロセス，構築サーバは実際にタスクを実行するプロセスである．

タスクの処理の流れは以下のようになる．(図 5.2)

1. ワークフロースケジューラは Makefile の解析結果から生成可能なタスクを生成し，分散スケジューラへシェルコマンドとして渡す．

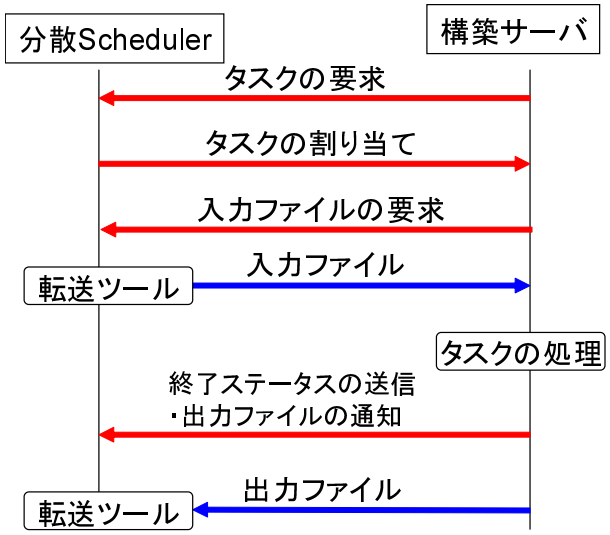

図 5.2: タスク処理の流れ

2. 分散スケジューラは構築サーバプールからサーバを選び出し，そのサーバへとコマンドと入力ファイルのリストを送る．
3. 構築サーバはその入力ファイルのリストの中からローカルにないファイルを分散スケジューラに要求する．
4. 分散スケジューラは要求されたファイルをファイル転送ツールへ伝える．
5. ファイル転送ツールはワークフロースケジューラのファイルシステムから構築サーバのファイルシステムへファイルを転送する．
6. 分散スケジューラはファイル転送ツールが正しく終了したことを検知すると，構築サーバへタスクを開始するよう伝える．
7. 構築サーバはコマンド (タスク) を実行する．
8. 構築サーバは終了ステータスと出力ファイルのリストを分散スケジューラに送る．
9. 分散スケジューラはその出力ファイルのリストをファイル転送ツールに伝える．
10. ファイル転送ツールは構築サーバのファイルシステムからワークフロースケジューラのファイルシステムへファイルを転送する．
11. 分散スケジューラはファイル転送ツールが正しく終了したことを検知すると，ワークフロースケジューラへタスクが終了したことを伝える．

5.2 ファイル転送ツール

5.2.1 利用方法と実装

ファイル転送ツールは通信に ssh または raw socket を用いてファイル転送を行うツールである．

```
1 gxpc mw xcp −−ssh=h1:h2 h1:path1 h2:path2
```

のように用いることで，ホスト h1，h2 間の通信に ssh を用いて，h1 の path1 を h2 の path2 へコピーすることができる．ssh の部分を socket にすることで ssh を使わない socket での通信になる．`--ssh` や`--sockeet` オプションを複数連ねることで，中間のホストで中継し，リスト状に転送を行うことも可能である．例えば’’`--ssh=h1:h3 --socket=h3:h2`’’ というように記述すると，h1 から h3 間では ssh での通信，h3 から h2 へは通常の socket での通信で，h3 で中継を行うことができる．

実装は以下のようになっている．各プロセスは gxpc mw を使ってエンドポイントを交換し，オプションで指定されたリストの次のホストへ connect する．オプションが socket の場合には socket を用いて直接，ssh の場合には’’`ssh HOST CMD`’’ で `HOST` 上で `CMD` が実行されることを利用して，リモートホストの `CMD` から connect を行う．このように接続したあと，ファイルを一定 (現在の実装では 1MB) ごとに分割して send する．中継ノードでは recv，send を繰り返すことで，全体としてはパイプライン状にファイル転送が行われる．

5.2.2 予備実験

ファイル転送ツールの性能予備実験として，スループットと CPU 使用率について測定を行った．使用したノードは hongo100，chiba100，chiba101 である．chiba100 と chiba101 は同一クラスタにあり，ノード間はギガビットイーサネットで接続されている．実験は 1GB のファイルを hongo100 から送り，chiba100 で中継を行い，chiba101 で受け取る．なお，hongo100 と chiba100 間の iperf[18] で測定したバンド幅は 780Mbps，scp でコピーした際のスループットは 65Mbps であった．

結果は表 5.1 のようになった．表の通信方法は順に，(hongo100，chiba100 間)/(chiba100，chiba101 間) に用いた通信方法である．まず，スループットについて述べる．ssh を用いた場合には scp の 1.2

表 5.1: 転送スループットと CPU 使用率

通信方法	転送スループット (Mbps)	CPU 使用率 (%)		
		hongo100	chiba100	chiba101
ssh/ssh	82	16	25	25
ssh/socket	78	16	25	17
socket/socket	682	18	97	95

倍程度であった．また，ssh を用いない場合には iperf で測定した hongo100，chiba100 間のバンド幅の 0.9 倍程度であった．これから，パイプライン転送を行う場合には，間に 1 ノードの中継を挟んでも，性能が 0.9 倍程度の劣化で済むことが分かった．また，ssh と socket を併用する場合には ssh で律速されてしまい，両方に ssh を使った場合と変わらないことが分かった．

次に CPU 使用率について述べる．表の中でも特に注目すべき値になっているのが socket/socket で通信した場合の chiba100，chiba101 の値である．いずれも 90 パーセント後半の値を示しており，非常に CPU 使用率が高い．

以上のことから，転送ツールはスループットに関しては十分な性能を示しているが，CPU 使用率が高くなっていることが分かった．CPU 使用率に関しては，実装を改善することで，改善の余地があると考えられる．

5.2.3 システム全体での位置付け

現在の実装では，GXP のルートノードから他のノードへのみしか転送が行えず，すべての出力ファイルは必ず 1 度ルートノードへ集められるようになっている．そのため，入出力ファイルのサイズが大きいタスクが多い場合には，ルートノードからの転送が性能ののボトルネックになる可能性がある．計算ノードから計算ノードへの直接のファイル転送については今後の課題とする．

分散スケジューラからのファイル転送要求はスレッド間同期キューを通して発行され，スレッドプール中のスレッドがそれを順に取り出して転送ツールを立ち上げるという実装になっている．

5.3 ワークフロースケジューラ

今回の実装では，ワークフロースケジューラとして Sun の dmake の機能の一部を用いた．dmake はオプションとして”-m grid” を与えると，SGE の qrsh コマンドを通して，タスクを投入する．qrsh は dmake から”qrsh *OPTION* /bin/sh *SCRIPT_FILE*” のような形式で実行されるので，qrsh と名前を付けたコマンドを実装することで，dmake の生成したスクリプトファイルを横取りすることが可能である．横取りしたスクリプトファイルを本システムで実行し，その終了ステータスに応じて，qrsh を終了させることで，dmake のほうではあたかもコマンドが SGE で正しく実行されているように見える．dmake は root ノードのみにインストールすればよいが，システムのインストールをより簡単に行えるようにするための独自のワークフロースケジューラを実装することは今後の課題とする．

5.4 dmake を使う上での実装上の工夫

dmake は実行できるタスクひとつについて，スクリプトファイルを 1 つ生成し，次のように qrsh を実行する．

```
1 qrsh −cwd −V −noshell −nostdin /bin/sh /home/enju/.dmake/dmake.script.kototoi000.17930.3.Y1jWew
```

この場合，/home/enju/.dmake/dmake.script.kototoi000.17930.3.Y1jWew がスクリプトファイルである．スクリプトファイルは，/bin/sh でそのまま実行できる形式になっている．dmake は qrsh の終了ステータスによって，タスクの成功，失敗を判断し，以降のフローの進め方をスケジューリングする．

5.4.1 入出力ファイルの抽出

dmake は dmake を実行するホストと構築サーバとのファイルシステムが共有されていることを前提としているため，入出力ファイルの情報は qrsh の方には渡されない．スクリプトファイルも，$@ や$¡などの Make のマクロは展開された状態である．そのため，本システムで利用するためには，入出力ファイルがどのファイルであるかということを，抽出する必要がある．今回の実装では以下のようにした．

5.4.1.1 入力ファイルの抽出

まず，スクリプトファイルをスペースで分割する．すると，ファイルやコマンド名やオプションなどの文字列に分けられる．その中で，ルートノードにファイルとして存在するような文字列を選び出す．これが，入力ファイルの候補である．ただし，中にはバイナリファイルなども含まれている．次に，構築サーバのほうで，これらの候補のなかで，存在しないファイルを選び出す．これらが，入力ファイルである．後は，これらのファイルを転送ツールによってルートノードから構築サーバのほうへコピーすると，ルートノードと構築サーバのファイル構成はまったく同じになるはずであるので，スクリプトファイルが実行できる状態になる．

5.4.1.2 出力ファイルの抽出

入力ファイルの抽出とほぼ同様に行う．スクリプトの実行が終了した時点で，スクリプト中に記述された文字列のうち，構築サーバにファイルとして存在するものを選ぶ．それらのファイルのうち，ルートノードに存在しないファイルが出力ファイルである．

5.4.2 スケーラビリティ

dmake はタスクひとつについて qrsh がひとつ立ち上がるという仕組みになっている．そのため，十分に多数のタスクがあるときには，ルートノードに構築サーバ数分 qrsh が立ち上がることになる．今回の実験の 800 個程度のオーダであれば，プロセスが 800 個立ち上がることは問題なかったが，qrsh からスクリプトファイルを分散スケジューラへ伝える部分に問題がおきた．

はじめの実装では，qrsh が立ち上がると，分散スケジューラへ UNIX Socket で接続し，スクリプトファイルの内容を送っていた．しかし，このような実装だと，socket 用のファイルディスクリプタを大量に開くことになるため，1 ユーザが開くことのできるファイルディスクリプタ数の制限 (デフォルトで 1024) に達してしまった．

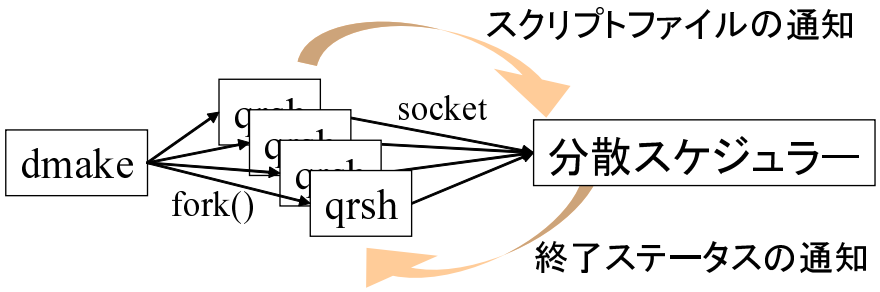

図 5.3: はじめの実装

そこで，今回の実装ではこれを避けるため，qrsh 自体は何もせず，分散スケジューラの側で，スクリプトファイルが生成されるディレクトリを定期的に監視して，スクリプトファイルを取得するという実装にした．このような実装では，qrsh からはファイルディスクリプタを一切開かないため，ファイルディスクリプタ数の制限に達することはない．また，qrsh のプログラム自体も小さくできるため，qrsh プロセスが大量に実行されても，800 個程度であれば問題なかった．(ただし，ルートノードとして利用しているマシンは，4 コア，メモリ 8GB を搭載したマシンであった．そのため，より一般的なマシンでの動作を確認する必要がある．)

しかし，このような実装の場合にも問題点がある．dmake は qrsh の終了をもって，タスクが終了したと判断するため，分散スケジューラ側から qrsh にタスクの終了を伝える手段が必要である (はじめの実装では，最初に開いたソケットを用いて通知していた)．これを解決するために，シグナルを用いることにした．シグナルには SIGUSR1，SIGUSR2 というユーザ定義が可能なシグナルが存在する．そのため，qrsh に SIGUSR1，SIGUSR2 のシグナルハンドラを定義し，SIGUSR1 だったら正常終了 (exit(0))，SIGUSR2 だったら異常終了 (exit(1)) とした．これによって dmake 側にタスクの成否を伝えることが可能である．qrsh のプロセス ID については，/proc/下の情報を調べることで，取得している．

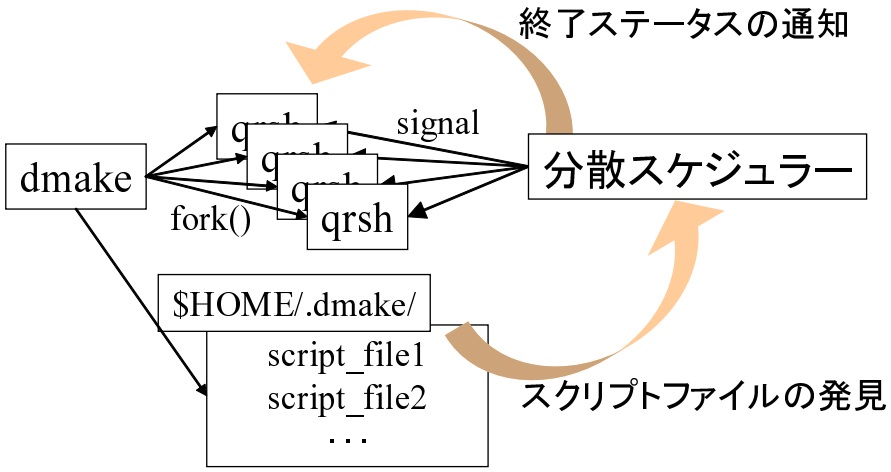

図 5.4: 改良を加えた実装

5.5 動的なタスクの生成

dmake では Makefile 中での dmake の呼び出し，すなわち，入れ子状の dmake をサポートしている．これによって，タスク中でタスクを生成し，同一のスケジューラによって，一緒にスケジューリングすることが可能である．これは例えば，file を並列化のために split を用いて分割するような場合に特に有効である．図 5.5 のような Makefile を作成すると，次のように実行が行われる．まず，1

```
1  output: input
2      split −a 3 −l 300 $< input.split.
3      $(MAKE) ‘echo input.split.∗ | sed −e ’s/in/out/’‘
4      cat output.split.∗ > $@
5  output.split.% : input.split.%
6      some_command $< > $@
```

図 5.5: 入れ子呼び出しを伴う Makefile

ノードにおいて input が 300 行ごとに分割されて，input.split.aaa，input.split.aab，. . . というファイルが作成される．次にマクロ$(MAKE) で再帰的に dmake が呼び出され，シェルのワイルドカードを使って展開することで，任意の数のファイルをターゲットとして与える．さらに，%を含む型ルールを用いることで，個々のファイルについて構築方法を示さなくても，任意の数のファイルが構築可能である．これによって，some_command が重い処理で，かつ，入力ファイルを分割可能な場合に，input を小さく分割することで，複数ノードで並列に処理を行うことが可能になる．

5.6 故障への対応

まず，ルートノードの故障に対しては，make と同様に，そこまで生成したファイルを元に，ワークフローを再開することができる．その他の計算ノードや，通信路が故障した場合については次のように対応する．GXP ではルートノード以外のプロセスが故障した場合には，通信ツリーにおいて故障したノードより下流のノードとの通信が行えない状態になる．つまり，割り当てたタスクが故障により終了しないのか，単にタスクのサイズが大きい・ノードの性能が低いのか区別が付かない．これに対応するには例えば，定期的に heartbeat を送るなどして能動的に故障を検知する方法も考えられる．しかし，本システムでは実装をなるべく単純化するために，そのような手法をとらず，次節に述べるタスクの再割り当てによってこれに対応する．すなわち，故障したノードに割り当てられていて，終了まで至らなかったタスクについては，再割り当てが行われ，別のノードで実行が行われる．

5.7 タスクの割り当てポリシー

ノードへのタスクの割り当ては，基本的に貪欲に割り当てていく．すなわち，タスク要求を受け取った順に現在実行可能なタスクを割り当てていく．さらに，故障や，極端に性能が低いノードが律速になってしまうことに対応するため，タスクの再割り当てを行う．

再割り当てのポリシーについては以下のものが考えられる．

1. 遊休ノードをなるべくなくすポリシー．すなわち，タスク要求に対して，現在あるタスクすぐさま再割り当てする．タスクが終了した場合には，タスクが終了したことを全体に知らせ，同じタスクを実行しているノードを停止する．冗長性が高くなるため，より高性能なノードでタスクが実行される可能性が高くなり，再割り当てを高速化に積極的に利用しているポリシーであるといえる．しかし，1 つのタスクが多数のタスクを生成するようなことが多いワークフローでは，再割り当てのタスクを実行をしているノードがいるにも関わらず，空いているノードがなく，未割り当てのタスクの実行が開始できないという場合が起きる可能性がある．

   予備実験を行った結果，今回のようなワークフローでは，未割り当てのタスクがあるにも関わらず，再割り当てのタスクが実行を始めてしまっているという上記のようなケースが頻繁に起きることが分かった．また，現在の実装ではワーカー数が 800 ノード前後になると，再割り当てのための入力ファイル転送がファイル転送要求用のキューにたまってしまい，すでにタスクが終了し出力ファイルの転送待ちの状態に関わらず，キューがいっぱいで，転送が始まらないということが起き，逆に遅くなってしまうこともあることが分かった．

2. 一定時間経っても，終了しないタスクについて再割り当てを行うポリシー．すなわち，故障や極端に遅いノードのみに対応するために再割り当てを行う．そのようなノードに割り当てられ

たタスクがワークフローのクリティカルパス上のタスクだった場合に，全体の性能が落ちてしまう可能性がある．

今回は，(2) のポリシーを採用した．

5.8 資源の増減

様々な資源利用ポリシーが現実には存在すると考えられるが，今回は特に，InTrigger[5] や東工大 TSUBAME クラスタなどで採用されているバッチキュー経由のプロセスの実行時間に制限があるような場合に対応するように実装を行った．ただし，1 タスクの粒度はプロセスの制限時間以内に実行が終わる大きさである必要がある．

タスクを実行するプロセスは，タスクを実行中であっても制限時間になると，分散スケジューラに途中終了したことを伝え，プロセスを終了する．ノードには GXP のプロセスが残るが，これは，GXP の機能を用いて GXP の通信ツリーから切り離し (gxpc trim)，終了させる．これで，バッチキュー経由で実行したプロセスを完全に終了することができる．一方，資源の増加については次のように対応する．一定時間間隔で，GXP で利用できそうなノードの獲得 (gxpc explore) を試みる．もし，新たにノードが獲得できた場合には，既存の mw と一緒に通信できるようにプロセスを立ち上げる (gxpc mw –join *CMD*).

第6章　実験

6.1　実験環境

実験環境として InTrigger プラットフォームおよび，研究科・研究室で所有するクラスタの計 9 クラスタを用いた．各クラスタの仕様は表 6.1 の通りである．ファイアウォール・NAT の様子については図 6.1 に示した．破線がファイアウォール・点線がプライベートアドレスを利用している範囲である．imade，kyoto に関しては，グローバルアドレスを持つノードは数ノードしかなく，残りのノードについては，プライベートアドレスが割り当てられ，外への通信には NAT を利用しているため，外から直接接続を張ることができない．また，kototoi のネットワークには外部からの接続に対し，ssh などの特定のポート以外にファイアウォールがかけられている．表 6.1 の上から 6 つのクラスタ (InTrigger プラットフォーム) についても各クラスタ間での通信は自由だが，外部からの接続に関してファイアウォールがかけられている．表 6.1 の上から 7 つのクラスタの計算ノードについては，ssh での利用も可能であるが，今回の実験では，様々な利用形態で利用可能なことを示すため，torque 経由で計算ノードにプロセスを立ち上げた．なお，使用コア数は，実験中に故障などで若干の変化があった．NLP モジュールはすべてユーザ権限で，各クラスタの NFS ファイルシステム上にインストールを行った． GXP のルートノードは kototoi に立ち上げ，そこから ssh で各クラスタ内の 1 ノードを獲得し，それらのノードからさらに torque/ssh で残りのノードを獲得した．kototoi のファイアウォールを回避するため転送ツールはすべて ssh 経由で行った．プライベートアドレスしか持たない imade，kyoto クラスタのノードに関しては，グローバルアドレスを持つノードで中継をしてファイル転送を行った．

6.2　実験をする上での細かな工夫

6.2.1　CPU アーキテクチャのヘテロ性への対応

chiba，hongo，istbs の各クラスタは，CPU アーキテクチャが i686 のマシンと，x86_64 のマシンが混在している．これは特に，ユーザが NFS で共有されたホームディレクトリにソフトウェアをインストールして利用する場合には，問題になる．i686 マシンでコンパイルしたバイナリは x86_64 マシンでも動作するが，逆は動作しない．そのため，そのようなクラスタでは，i686 マシンでコンパイルしたバイナリを利用するのが基本になる．しかし，x86_64 マシンにおいて，i686 マシン用のバ

表 6.1: 各クラスタの仕様

クラスタ名	設置場所	‘arch‘	使用コア数	ネットワーク	利用形態
chiba	国立情報学研究所	i686 & x86_64	58 & 116	グローバル	Torque
hongo	東京大学 (本郷)	i686 & x86_64	62 & 28	グローバル	Torque
imade	京都大学	x86_64	59	プライベート	Torque
kyoto	京都大学	x86_64	67	プライベート	Torque
okubo	早稲田大学	x86_64	27	グローバル	Torque
suzuk	東京工業大学	x86_64	71	グローバル	Torque
kototoi	東京大学 (本郷)	x86_64	42	グローバル (firewall)	Torque
istbs	東京大学 (本郷)	i686 & x86_64	89 & 77	グローバル	ssh
sheep	東京大学 (柏)	i686	65	グローバル	ssh

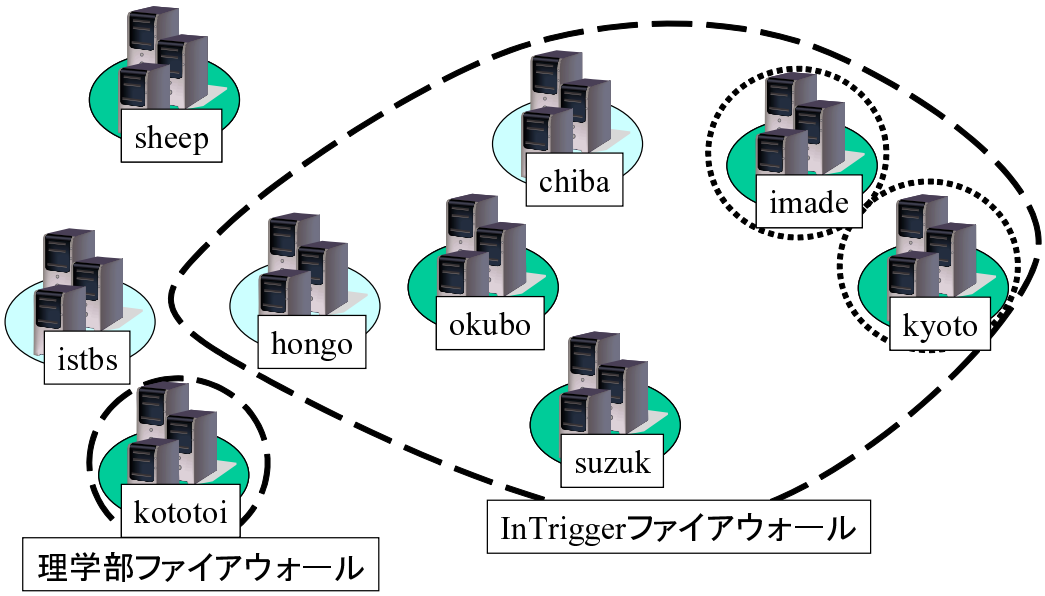

図 6.1: クラスタのファイアウォール・NAT の様子

イナリと x86_64 マシン用バイナリで性能を比較すると，一般に x86_64 マシン用にコンパイルしたバイナリの方が性能がよい．特に，実行時間が長くなるようなプログラムでは，実行時間の差が大きくなってしまう．

そのため，今回の実験では，各ホームディレクトリに i686 と x86_64 というディレクトリを作成し，各々に NLP モジュールをインストールした．また，実行されるときにも，各マシンのアーキテクチャにあわせて実行されるよう，Makefile に

```
1 $(OUTPUTFILES): $(INPUTFILES)
2         ARCH=‘arch‘ \
3         PROGDIR=$$HOME/$$ARCH \
4         $$PROGDIR/bin/binary $<
```

のようにして，環境変数を Makefile ではなく，リモートのホストで展開されるように記述し，各環境に合わせたバイナリが実行されるようにした．

6.2.2 測定値の取得

本実験では，各ノード CPU 使用率を測定しているが，このような大規模でヘテロな環境ではそのような測定自体，難しいことである．そこで，そのような測定値の取得に，VGXP[24] を用いた．VGXP は多数のマシンの状態をリアルタイムに表示するモニタリングシステムである．通常の利用では，java を用いたグラフィカルなインターフェースで情報を表示するが，そのようなクライアントと，マシンの情報を収集，送信するサーバとの通信はプレインテキストで行われている．今回は，情報をディスクに書き溜める VGXP クライアントを実装し，それを用いて CPU 使用率を取得している．

6.3 スケーラビリティに関する実験

SAT ソルバである zchaff を用いて，複数の SAT 問題を解くという実験を行った．TIMEOUT として 10 秒を設定した．実験は hongo クラスタを用いた．

この問題はワークフローではないが，make を用いて簡単に記述ができる．

```
1 # −∗− Makefile −∗−
2 %.res : %.cnf
3         $(ZCHAFF) $< 10 > $@
```

実行は，カレントディレクトリに複数の.cnf ファイルがあるとして，

```
1 $ gxpc make ‘echo ∗.cnf | sed‘ s/\.cnf/.res/g ’
```

のように行える．

6.3.0.1 dmake との比較

829 ファイルの SAT 問題について，どの程度並列化によって速度向上が得られたか dmake と比較を行った．なお，本システムに関しては NFS によって入出力ファイルをやり取りした場合と，ファイル転送ツールを用いてやり取りした場合とについて測定した．結果は図 6.2 のようになった．

まず，dmake と NFS を用いた本システムとを比べると，本システムのほうが若干よい性能を示した．これは，本システムでは，gxp によりあらかじめ通信路が確保されているのに対して，dmake ではタスクを実行する度に rsh によって，プロセスを立ち上げていることによると考えられる．

次に，ファイル転送ツールを用いた本システムは 64 ノードのときにも 12 倍程度の速度向上しか得られず，スケールしていない．これは，TIMEOUT が 10 秒であってもそれ以下の時間で解けてしまう問題がほとんどであり，タスクの粒度がファイル転送にかかる時間に比べて，小さかったことが大きな原因であると考えられる．

6.3.0.2 タスクの粒度とスケーラビリティ

次に，タスクの粒度を変化させたときに，速度向上はどのように変わるかを実験した．タスク数は 160 ファイルで，1 つのタスクは SAT 問題を解くのに加え，一定時間の sleep を行うようにした．sleep は 0 秒，10 秒，20 秒，30 秒に設定し，それぞれ測定を行った．結果は図 6.2 のようになった．

sleep の時間を大きくするほど，すなわち，タスクの粒度を大きくするほど，スケールしていることが分かる．64 ノードで若干グラフの傾きが小さくなっているのは，タスク数が少なすぎたために (1 ノード平均 2.5 個のタスク)，終了前の時間でアイドルなノードが増えてしまったためであると考えられる．すなわち，もし，ファイル転送がなく，すべてのノードが正確に同じ性能であったとすると，2 回タスクを割り当てられるノードと 3 回割り当てられるノードがあり，それぞれ半数ずつである．よって，半数のノードは 2/3 の時間はアイドルであることになる．

6.4 自然言語処理ワークフローの実行

MEDIE/Info-Pubmed データベースを作成する Makefile を本システムを用いて実行した．

6.4.1 ワークフローの概要

実験で使用した Makefile はコメント，空行なしの行数にして 414 行，ターゲット数 54 の大きな Makefile である．図 6.4 は拡張子によってファイルの依存関係を示したワークフローである．各ノードがファイルで，矢印の向きに依存関係がある．数字はそれぞれ，

1. Pubmed からリリースされた xml からアブストラクトを抜き出す

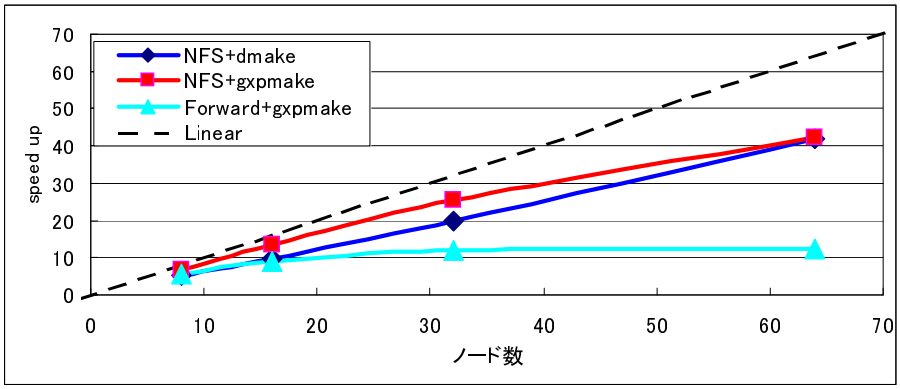

図 6.2: ノード数を変化させたときの速度向上

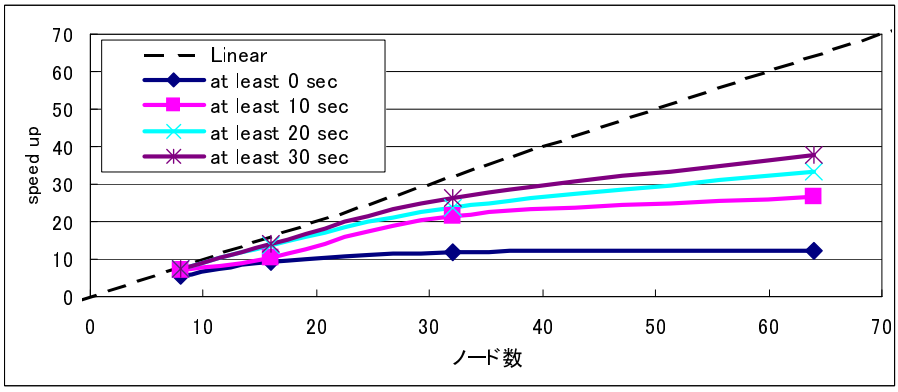

図 6.3: ノード数とタスクの粒度を変化させたときの速度向上

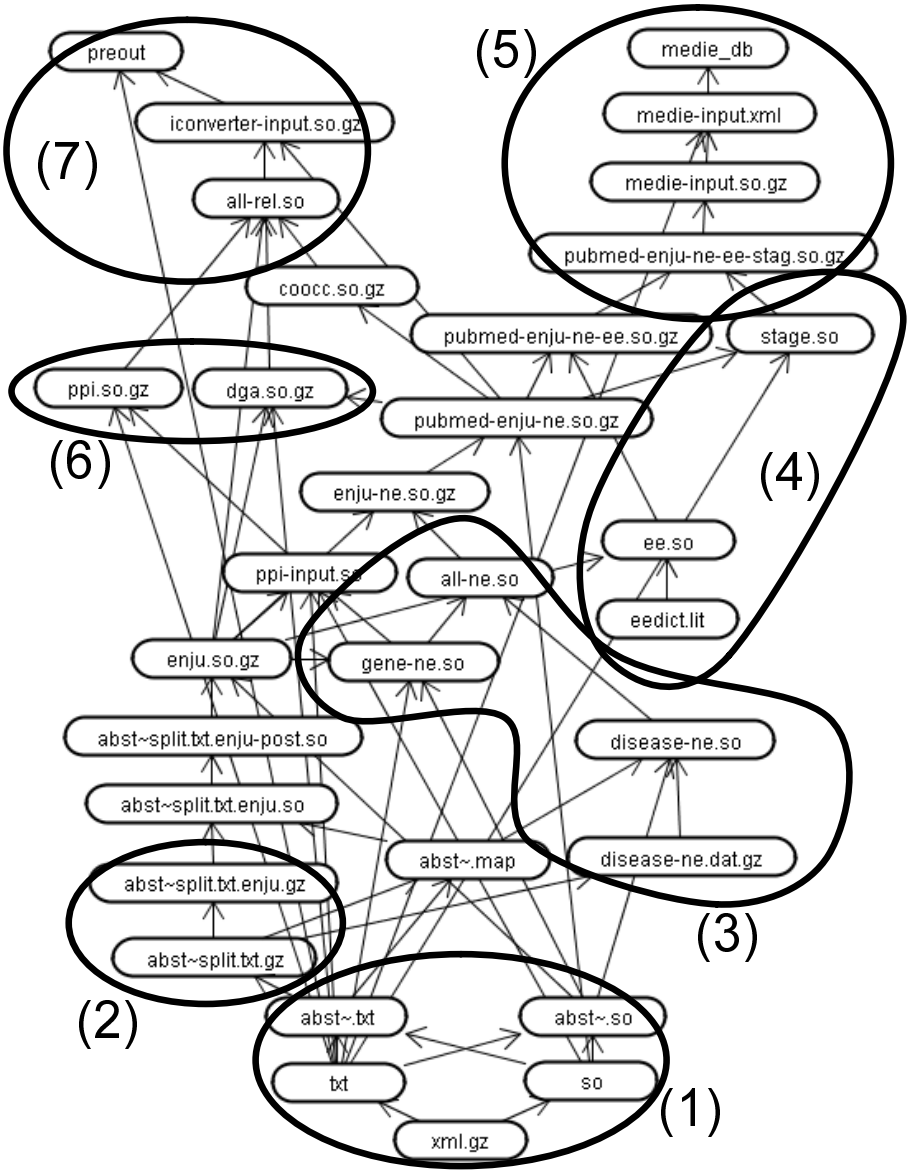

図 6.4: 拡張子で示したワークフロー

2. アブストラクトを構文解析する
3. 遺伝子・疾患に関する固有表現を抽出
4. イベント表現・文のタイプにタグ付け
5. MEDIE 用の xml を作成
6. たんぱく質の相互作用・遺伝子と疾患の関連を抽出
7. Info-Pubmed 用のデータを作成

のような処理を行っている．全体の処理の上で最も処理に時間がかかるのが，HPSG 構文解析 (enju) の部分であり，ファイルを分割してマシンごとに分散して実行し，後で結合する．

6.5 実験結果

今回の実験では，enju の入力ファイルを 1000 行ごとにファイルを分割した．入力ファイルとして，medline07n0400.xml.gz から medline07n0419.xml.gz の計 20 ファイル用意し，それぞれに対してデータベースファイルを構築した．各ファイルには enju の入力になる文章が 30000 ずつ含まれている．20 ファイル合計で行数は 4,642,430 行である．

一度に多量のタスクが生成されてしまわないよう，各ファイルは 400 秒 (6 分 40 秒) ごとの間を開けて，ワークフロースケジューラに渡した．また，タスクの再割り当ては前回割り当ててから 50 分経つごとに行った．図 6.5 はシステムが起動してから，48 分間の各クラスタの CPU 使用率を合計したものである (ただし，chiba，hongo，istbs，kototoi，sheep については，測定時間中に他のユーザの利用があった)．NLP のモジュールは実行中ほぼ CPU を 100 パーセント使い切るので，CPU 使用率=利用コア数と考えても問題ない．開始 16 分のところで急激に利用コア数が増加している．これは，enju 用の入力ファイルまでワークフローが進み，複数のコアで enju が開始されたためである．分割前の 1 つのファイルには約 230000 行程度の文章が含まれているので，タスク数にして約 230 個の enju 実行タスクが一度に生成される．開始 16 分の時点では，0 秒と 400 秒に開始されたファイルが enju の入力ファイルを生成し，500 程度のノードで分散してタスクを開始している．その後，20 分前後と 25 分前後に同様に enju 実行タスクが生成され，タスクがリソースに割り当てられている．28 分の時点ではほぼすべての利用可能なリソースを使い切ることができている．グラフの立ち上がりから見ると，タスクが生成されてから 1 分程度でほぼ 200 タスクが開始されている．

enju の出力ファイル (合計 4643 個) が生成された時刻のヒストグラムは図 6.6 のようになった．400-403 番のファイルはほとんどが 1 時間 20 分以内に実行を終えている．グラフでは確認できないが，3 時間した経過したあたりにも作成されたファイルがあり，なんらかの理由によって失敗したタスクが，再割り当てによって実行されたと考えられる．enju のタスクがすべて終了したのが開始後

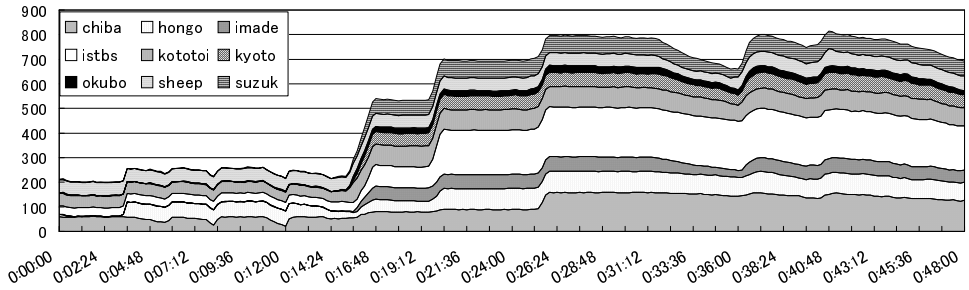

図 6.5: CPU 使用率の変化

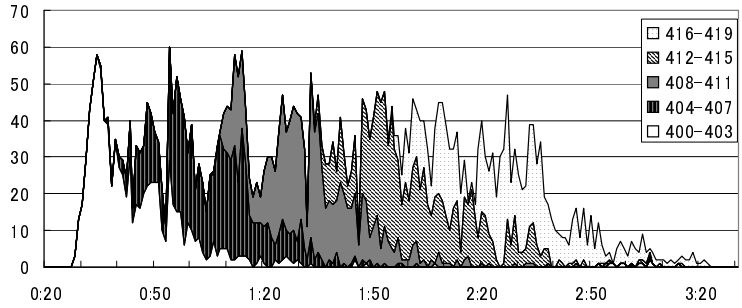

図 6.6: enju の出力ファイルの作成時刻のヒストグラム

表 6.2: enju の処理ファイル数と平均実行時間

	chiba0	chiba1	hongo0	hongo1	imade	istbs0
処理タスク数	231	786	337	189	381	248
平均処理時間	18:44	13:54	18:49	13:56	13:56	43:18

	istbs1	kototo	kyoto	okubo	sheep	suzuk
処理タスク数	605	512	440	200	204	510
平均処理時間	26:54	12:32	14:33	13:51	33:22	13:55

3 時間 20 分後であった．また，enju の実行時間の測定値から全コアの平均的なファイルの処理時間は 1264sec で，全体で 4643 ファイル，全参加コア数は 761 コアであったことから，およその並列化効率は

$$(T_1/T_n)/n = (1264 * 4643/12000)/761 = 0.64$$

であった．ただし，この数字は，ノードを獲得したり，enju のタスクを生成するまでの時間も含んでいるため，enju のタスクのみにかかった時間から算出するものに比べ，小さめになっている．

これらの結果から，本システムはワークフロー上でもっとも計算時間を必要とする HPSG 構文解析を，開始 30 分ころにはリソースを使い切る程度に効率よく並列に実行できていることが分かった．また，再割り当ての仕組みが正しく動作していることが分かった．しかしながら，Embarrassingly parallel なタスクとしては並列化効率が低く，終了間際に遅いノードや，なんらかの理由で再実行されることになったタスクが全体の性能を引き下げていると考えられる．

第7章　おわりに

7.1　まとめ

本論文では Make をヘテロな分散環境で実行するためのシステムを設計・実装した．ヘテロな環境とは，ネットワーク (ファイアウォール・グローバルアドレスの有無) や，利用形態 (ssh などによる直接ログインしてインタラクティブに利用可能か否か，バッチキュー経由で利用するか) が不均一であるような環境である．そのような環境で利用するために，分散プロセスマネージャとして GXP を用いた．GXP は多様な利用形態に対応し，プロセスを立ち上げることができる．

また，資源の増減・故障に対応するため，GXP を拡張しプロセスを途中参加できるようにした．この機能を用いることで，ルートノードさえ故障しなければ，利用する資源を次々に変更しながら計算を続行することが可能になった．

また，そのような環境で利用するためのファイル転送システムを設計・実装した．これは，通信路として通常の socket のほか，ssh も利用できるようにすることで，ファイアウォールによる制限を回避しやすい．さらに，複数のノードを中継してパイプライン上にファイルの転送を行える機能を実装し，プライベートアドレスしか持たないようなノードにも，ログインノードを中継することで直接ファイルを転送できるようにした．

実験では，実環境で Web サービスである MEDIE と Info-Pubmed のためのデータベースの構築を行った．実験環境は，複数の管理サイトに別れ，複雑なファイアウォールや利用形態のある実際にヘテロな環境であった．また，データベース構築の複雑なワークフローが記述された Makefile を処理した．中でも構文解析の部分では並列化効率が 64%であることを示し，システムはまだ改善の余地があることが分かった．

7.2　今後の課題

今後の課題としては，

- ファイルの転送をルートノードを介さずに行うことで，出力ファイルが大きな場合にルートノードがボトルネックにならないようにする
- タスクの再割り当てのポリシーを変更し，より並列化効率が高くなるようにする

- dmake に依存した部分を，独自のものに書き換える
- システムを実際に利用してもらえるようなソフトウェアとして公開できる程度に頑健に書き直す

などがあげられる．

参考文献

[1] Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman/.

[2] Enju - A practical HPSG parser. http://www-tsujii.is.s.u-tokyo.ac.jp/enju/.

[3] GXP : Grid and Cluster Shell. http://www.logos.ic.i.u-tokyo.ac.jp/gxp/.

[4] Info-PubMed. http://www-tsujii.is.s.u-tokyo.ac.jp/info-pubmed/.

[5] InTrigger プラットフォーム. https://www.logos.ic.i.u-tokyo.ac.jp/intrigger/.

[6] MEDIE. http://www-tsujii.is.s.u-tokyo.ac.jp/medie/index.html.

[7] MPICH Home Page. http://www-unix.mcs.anl.gov/mpi/mpich1/.

[8] Sun WorkShop TeamWare ユーザーズガイド. http://docs.sun.com/app/docs/doc/806-4846.

[9] K. Amin, G. von Laszewski, M. Hategan, NJ Zaluzec, S. Hampton, and A. Rossi. GridAnt: a client-controllable grid workflow system. *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 210–219, 2004.

[10] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. *Grid Computing: Second European AcrossGrids Conference, AxGrids 2004, Nicosia, Cyprus, January 28-30, 2004: Revised Papers*, 2004.

[11] I. Foster and C. Kesselman. Globus: a Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, 11(2):115, 1997.

[12] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[13] Matthew E. Hoskins. Sshfs: super easy file access over ssh. *Linux J.*, 2006(146):4, 2006.

[14] G. Laszewski, I.T. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):645–662, 2001.

[15] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows, 2004.

[16] Kenjiro Taura. GXP: An Interactive Shell for the Grid Environment. *Proc. IWIA2004*, 1:59–67, 2004.

[17] Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix : a parallel programming model for accommodating dynamically joining/leaving resources. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, pages 216–229, 2003.

[18] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf-The TCP/UDP bandwidth measurement tool. *URL: http://dast. nlanr. net/Projects/Iperf*, 2004.

[19] 関谷 岳史, 田浦 健次朗, and 近山 隆. 適応的並列計算を支援するプロトコルの設計と正当性の証明. **先進的計算基盤システムシンポジウム** *SACSIS 2007*, pages 349–358, 2007.

[20] 建部 修見, 森田 洋平, 松岡 聡, 関口 智嗣, and 曽田 哲之. ペタバイトスケールデータインテンシブコンピューティングのための Grid Datafarm アーキテクチャ. **情報処理学会論文誌*: ハイパフォーマンスコンピューティングシステム***, 43:184–195, 2002.

[21] 青木 仁志, 中田 秀基, 田中 康司, and 松岡 聡. 動的なノード群構成機構を備えた階層型グリッド環境: Jojo2. **先進的計算基盤システムシンポジウム** *SACSIS 2006*, pages 101–108, 2006.

[22] 大田 朋子. 自然言語処理技術に基づく意味構造を利用した情報検索と情報抽出. **情報管理**, 49(10):555–563, 2007.

[23] 堀田 勇樹, 田浦 健次朗, and 近山 隆. 耐故障並列計算を支援する自律的な故障検知機構. *IPSJ Transactions on Advanced Computing Systems*, 46(SIG 12):236–244, 2005.

[24] 鴨志田 良和, 金田 憲二, 遠藤 敏夫, 田浦 健次朗, and 近山 隆. 低負荷で多数の計算機をリアルタイムに監視するシステム VGXP の実装. **電子情報通信学会技術研究報告**. *CPSY,* **コンピュータシステム**, 106(199):19–24, 2006.

発表文献

1. 関谷 岳史，田浦 健次朗．グリッド用シェル GXP の長時間計算のための拡張．並列／分散／協調処理に関するサマーワークショップ（SWoPP2007），pp.297-302，旭川，2007 年 7 月．

2. 関谷 岳史，田浦 健次朗，近山 隆．適応的並列計算を支援するプロトコルの設計と正当性の証明．先進的計算基盤システムシンポジウム（SACSIS 2007），pp.349-358，東京，2007 年 5 月．

3. 関谷 岳史，田浦 健次朗，近山 隆．適応的並列計算を支援するプロトコルの設計と正当性の証明．並列／分散／協調処理に関するサマーワークショップ（SWoPP2006），pp.169-174，高知，2006 年 7 月．

4. 斎藤 秀雄，鴨志田 良和，澤井 省吾，弘中 健，高橋 慧，関谷 岳史，頓 楠，柴田 剛志，横山 大作，田浦 健次朗．InTrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境．情報処理学会研究報告 HPC-111 (SWoPP 2007)，pp.237-242，旭川，2007 年 8 月．

5. 鴨志田 良和，斎藤 秀雄，横山 大作，弘中 健，澤井 省吾，高橋 慧，関谷 岳史，柴田 剛志，田浦 健次朗．構成変化への柔軟な対応を考慮した他拠点にわたる分散計算機環境構築．先進的計算基盤システムシンポジウム (SACSIS 2007)，東京，2007 年 5 月（ポスター発表）．

謝辞

本研究を進めるにあたり，近山隆教授ならびに田浦健次朗准教授には，貴重な指摘，助言をいただきました．近山先生には，特に発表練習のときに，新しい視点を与えてくれるようなアドバイスを多くいただきました．田浦先生には，本当にいろいろとお世話になりました．研究の進め方から，プログラムの書き方，python のすばらしさなど教えていただきました．先生に熱心にご指導いただき，研究のペースを作っていただいたおかげで，この論文を仕上げることができたと思います．本当にありがとうございました．

助教の横山大作さんをはじめ，研究室の先輩の鴨志田良和さん，斎藤秀雄さん，白井達也さん，高橋慧さんには技術的なことはもとより，いろいろと教えていただき，特にお世話になりました．また，気さくに接していただき，くじけそうになることもある研究生活も楽しく過ごすことができました．中でも斎藤さんには卒論生の頃から，TA の活動を含め，本当にお世話になりました．そのほか，卒論のころから一緒にやってきた同級生の斎藤大君，坂本祥吾君，吉田慎一郎君や，弘中健君をはじめとした優秀な後輩にも，色々と助けていただきました．みなさんと研究生活が送れて本当によかったです．ありがとうございました．

本研究の実験をするにあたり，辻井潤一教授ならびに助教の松崎拓也さんをはじめとした辻井研究室のみなさまに，アドバイスをいただき，また，NLP のモジュールを提供していただきました．また，各クラスタを構築・管理していただいた方々のおかげで，非常に大規模な実験をすることができました．ありがとうございました．

ここには書ききれませんが，本当に多くの方にお世話になったと思います．皆様本当にありがとうございました．

平成 20 年　2 月　4 日