社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－HPC－111 (51)
2007／8／3

グリッド用シェルGXPの長時間計算のための拡張

関谷 岳史† 田浦 健次朗†

GXPはユーザ権限で簡単にインストールが可能なグリッド用のツールである．また，GXPはプロセス間の通信をローカルのファイルディスクリプタへの読み書きで行え，簡単に並列プログラムを作成することをサポートしている．本研究では，GXPを拡張しInTriggerプラットフォームなどの複数クラスタ環境で大規模長時間計算を行うために必要となる機能を追加した．ひとつは，故障したノードの検知とその情報の各ノードへの伝達，もうひとつは，計算実行中のノードの追加獲得と，それによって獲得したノードへのコマンドの投入である．また，実装による実環境での性能評価を行い，拡張機能が実用上十分な速度で動作することを示した．

An Extention to Grid-Shell GXP for Persisting Computation

Takeshi Sekiya† and Kenjiro Taura†

GXP is a tool that can be installed easily by user authority and can be used in a grid environment. GXP aids easy development of parallel programs by allowing nodes to communicate by simply reading and writing local file descriptors. In this research, we enhanced GXP and added functions needed for large scale and persisting computation in multi-cluster environments like the InTrigger platform. One is fault detection and spreading this information to each node. The other is the acquirement of new nodes while other nodes are executing calculation and starting command in these new nodes. We tested the performance of this implementation in a real environment and we demonstrate that this implementation can sufficiently in practice.

1. はじめに

1.1 背景

InTriggerプラットフォーム[1]に代表されるような，複数のドメインにまたがったクラスタ同士をつなげた大規模な計算資源を利用できる機会が増えている．しかし，単一のクラスタ環境に比べて，これらの環境は利用するのが難しい．それは，このようなグリッド環境のヘテロ性，つまり，ドメイン間でのハードウェア，ネットワーク，OS，ミドルウェアなどの違いに起因する場合がほとんどである．

このような環境を便利に利用することを目指した研究もたくさん行われている．ただし，それらの多くは，管理者によって，すべてのドメインに同一なソフトウェアをインストール・設定する必要がある場合がほとんどであり，導入時のコストやメンテナンスなど管理コストが大きくなりがちである．また，より大規模な計算資源を利用しようとすれば，複数の管理ドメインに属する資源を使う必要が出てくる．そのような場合には，そもそも使いたいソフトウェアがインストールされていない可能性もある．

このような問題に応えるソフトウェアのひとつとして，GXP[2]がある．GXPは最小限の基盤ソフトウェアのみで動作し，ユーザ権限で簡単にインストールができるグリッド用のツールである．GXPでは決まった番号のファイルディスクリプタを読み書きするだけで，プロセス間の通信が可能になるため，簡単に並列計算のプログラムを書くことができる．

1.2 本研究の目的

本論文では，GXPの紹介と，GXPを使って長時間大規模計算を行うための拡張について述べる．特に，長時間の実行の際には必要になると考えられる，故障が起きた場合の対処や，計算実行中に資源を新しく獲得するための拡張を行う．

本論文の構成は以下の通りである．2節で関連研究を紹介する．3節でGXPについて紹介し，4節で今回行った拡張について述べる．また，5節で今回行った実験とその結果を示す．

† 東京大学
University of Tokyo

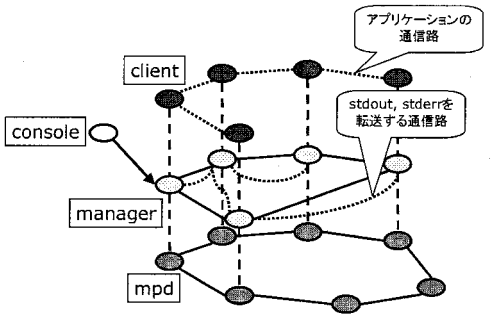

図 1　MPD の構成

2. 関連研究

2.1 MPD

MPD[3)4)] は MPICH で利用されているプロセスマネージャである．**図 1** が MPD の構成を示したものである．図 1 中の mpd は，もっとも下のレイヤにあたり，ユーザからの認証を受け付けて上位の manager プロセスを立ち上げる役割を果たす．mpd はユーザ権限でも実行が可能であるが，基本的には管理者権限の実行を想定しており，ノードの起動中は常に実行されていることが仮定されている．また，mpd 同士の接続はリング構造になっている．ノードの故障が起きた際には，自動的にそれが検出され，リングが再構成されるようになっている．

manager は client(アプリケーション) プロセスを管理するためのプロセスである．これは，console からの命令によって，client プロセスにシグナルを届けたり，各 client の stdout/stderr を集めて，console に届けるという役割を果たす．各 manager は MPD から他の manager と接続するための情報を受け取り，manager 同士でリング構造の通信路を構築する．また，パフォーマンスの向上のために stdout/stderr を転送するための通信路をこれとは別に，木構造になるように確立する．

client はアプリケーションプロセスで，MPI のプロセスであることが多いが，他のアプリケーションでもよい．たとえば，5) では MPD を利用して並列 Unix コマンドを実装している．

MPD はユーザ権限でも実行はできるものの，基本的には管理者権限でのインストールが想定されている．また，グリッド環境で利用するためには/etc/hosts の編集が必要であり，管理者権限および，同一管理ドメインであることが必要になってしまう．

2.2 GXP の利用例

MC-MPI[6)] はマルチクラスタ環境を想定した MPI ライブラリである．MC-MPI はメッセージをソフトウェアのレベルで転送することで，NAT やファイアウォールのある環境でも使用が可能である．また，ネットワークの遅延の測定と，アプリケーションの通信をプロファイルして，確立する接続を決めることで，少ない接続数で高い通信性能を示している．MC-MPI では MPI プロセスの起動と，各プロセスのエンドポイントの交換のために GXP を利用している．MC-MPI において GXP は MPICH でいうところの MPD の役割を果たしていると考えられる．

VGXP[7)] は複数のクラスタをモニタリングするためのソフトウェアである．複数ノードのシステム情報を集めて可視化することで，クラスタの混雑状況の監視や，並列プログラムの開発に役立てることができる．VGXP でも監視に必要なプロセスの起動に GXP が利用されている．さらに，VGXP から GXP のコマンドを実行する機能もサポートしている．

他にも，GXP は言語処理の分野で大規模なプログラムを並列化するのにも利用されている (8) など)．

3. GXP

3.1 GXP の構成

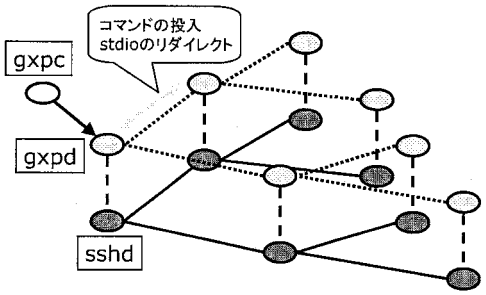

図 2　GXP の構成

GXP の構成は**図 2** のようになっている．GXP のプロセスとしては 2 種類のものがあり，ひとつは gxpc，もうひとつは gxpd である．gxpc はコマンド実行のたびにユーザによって生成されるプロセスであり，ユーザは gxpc に引数として命令を与える．gxpd は gxpc の最初の実行の際に起動され，gxpc quit されるまで，ずっと走り続ける．gxpd は主に (1) 子プロセスを起動する (2)gxpc の stdin を子プロセスへリダイレクトする (3) 子プロセスの stdout/stderr を gxpc へリダイレクトするという 3 つの機能を持つ．図 2 では最

も下の通信レイヤとしてsshdを利用しているが，現在の実装ではrshやtorqueなども利用できるようになっている．

3.2 MPDとの比較

GXPはMPDとよく似た機能を提供するが，構成上は様々な違いがある．GXPはMPDと違い，最も下の通信レイヤとして，sshなどの一般的に利用されているシステムを用いている．そのため，管理者権限による特別なソフトウェアのインストールや設定が必要ない．また，一般的に利用されているシステムなので，daemon自身のバグによって，daemonが利用できなくなってしまう可能性も低い．

MPDがリング構造で接続をするのに対し，GXPは木構造に接続を張る．これは，ファンアウト数だけ並列にノードの獲得を行うことができるため，gxpdを立ち上げるのを高速に行うことができる．また，例えば，外部からログインノードを介してのみ，計算ノードにログインできる，というようなクラスタを利用する際には，木構造にならざるを得ない．一方，leafノード以外のノードが故障した際には，故障したノードより下流のノードもrootノードと通信ができなくなってしまうという欠点もある．

3.3 GXPの利用

GXPを利用するためのに必要な環境は，sshで接続が可能な標準的なLinuxのシステムだけである．GXPはpythonスクリプトで書かれているため，それ自体のインストールもただファイルをコピーしてくるだけでよい．また，GXPは新しいノードを獲得する際，自分自身を自動的に新しいノードにインストールするため，ユーザはgxpcコマンドを実行するノードにのみGXPをインストールしておけば十分である．

ここで，GXPの利用例を示す．なお，プロンプト**[x/y/z]%**の意味は全部でzノードが獲得されているうち，yノードでコマンドを実行し，xノードが直前のコマンドを成功したということを表している．

3.3.1 ノードの獲得・選択

まず，ノードを獲得するためにどのようなログイン方法を用いるかを指定する．

```
[1/1/1]% gxpc use ssh hongo000 okubo000
[1/1/1]% gxpc use ssh okubo
```

ホストhongo000でgxpcを実行したものとする．1行目はhongo000からokubo000へログインするためにsshを用いることを指定している．2行目はokuboと付くホストの間では同様にsshを用いることを指定する．

```
[1/1/1]% gxpc explore okubo000
reached : okubo000
[2/2/2]% gxpc explore 'okubo[[001-013]]'
reached : okubo001
reached : okubo002
...
reached : okubo011
[15/15/15]%
```

1行目ではホストokubo000の獲得を試み，2行目で獲得できたというメッセージが表示されている．同様に，3行目では，okubo001からokubo013へログインを試みており，4行目以降で獲得できたというメッセージが表示されている(表示順は必ずしも数字の順ではない)．

ノードの選択を行うには，まず，各ノードでコマンドを実行し，終了ステータスの違いで，選択するノードを分ける．

```
[15/15/15]% gxpc e 'hostname |grep okubo'
okubo010
...
okubo000
[14/15/15]% gxpc smask
[14/14/15]%
```

例えば，1行目で各ノードで**hostname |grep okubo**を実行し，ホスト名に'okubo'を含むもので，コマンドが成功する．下から2行目のプロンプトの左端の数字で，14ノードでコマンドが成功したことを示している．このあとで，smaskというコマンドを実行すると，直前のコマンドが成功したノードのみを選択することができ，以降のeコマンドは選択されたノードでのみ実行されることになる．

3.3.2 eコマンド

eコマンドは現在選択されている各ノードでコマンドを実行するためのコマンドである．デフォルトではeコマンドへのstdinが各ノードで実行されるコマンドのstdinにリダイレクトされ，各ノードで実行されるコマンドのstdout/stderrがeコマンドのstdout/stderrにリダイレクトされる．

さらにeコマンドではオプション次第で，ファイルディスクリプタをリダイレクトする機能が利用できる．例えば，-updown 3:4というオプションをつけると，各ノードのファイルディスクリプタ3への出力が，各ノードのファイルディスクリプタ4へ入力される．

これを用いることで，例えば以下のように簡単なマスターワーカのプログラムを作成することが可能である．

```
### master_worker.py ###
```

```
#!/usr/bin/env python
import os
idx = os.environ["GXP_EXEC_IDX"]
if idx == 0:
  while 1:
    (タスク要求のメッセージが来るまで FD3 を読む)
    タスクの生成
    (タスクと処理させるワーカの ID をのせたメッ
    セージを FD4 へ書き込む)
else:
  while 1:
    (自分宛のタスクが来るまで FD3 を読む)
    タスクの実行
    (タスク要求のメッセージを FD4 へ書き込む))
### master_worker.py ###

 % gxpc e --updown 3:4 ./master_worker.py
```

e コマンドが実行される各ノードの環境変数 GXP_EXEC_IDX にはその実行において一意な ID が設定されている．このプログラムでは，ID 0 番をマスタープロセス，それ以外をワーカプロセスとして利用することになる．

4. GXP の拡張

本研究では，前節で紹介した e コマンドを用いて長時間の計算を行うための拡張を行う．長時間にわたる大規模計算の実行に必要なことは以下のようなものがあげられる．

(1) (耐故障性) 並列計算を行う場合にはノードの数に比例して，故障の確率が高まる．また，長時間にわたる計算の場合には，故障の度に始めからやりなおすわけにはいかないので，故障に対するなんらかの対応が必要になる．

(2) (計算資源の増減) 複数の利用者によって共有されている資源の場合，ひとりで占有することは難しく，仮に占有できたとしても，時間制限があることもある．また，バッチキューによる資源管理を行っている場合，キューが混んでいるための順番待ちですぐに実行が開始できない可能性もある．複数の管理ドメインの資源をまとめて利用したい場合には，このような理由により，利用可能な時間がばらばらである可能性が高い．そのため，計算実行中にプロセスの参加や脱退が行えるようにするなどの，柔軟な資源管理が求められる．

本研究ではこの内，耐故障に関する拡張と，計算実行中の資源の追加について実装を行った．

4.1 故障の検知

今回の実装では，heartbeat メッセージを一定間隔で送ることで，故障を検知する手法を用いた．これは，gxpc から順に下流のほうへ木構造に沿って ping メッセージを送り，各ノードが自分の子ノードの生存を確認し，各ノードは自分と自分の子ノードの情報を再びルートノードへ届けるというものである．gxpc が故障を検知すると，その情報を各プロセスへブロードキャストし，各プロセスは標準入力からその旨を読み出すことができるように実装を行った．

このような実装では，各プロセスからの返事がすべてルートノードに集まることになる．heartbeat 間隔を T_{HB} とし，gxpd の数を n ノードとすると，ルートノードの gxpd では単位時間当たり n/T_{HB} 個のメッセージを受け取る．よって，n が大きくなってくると，ルートノードに負荷が集中し，処理のボトルネックになる可能性がある．この問題への対応は今後の課題とするが，各ノードからの返事のメッセージをまとめて，メッセージ数を減らすなどの対応が考えられる．

4.2 e コマンド実行中のノードの獲得

本拡張では，e コマンドの実行中にノードを獲得できるように実装を行った．具体的には，プロセスの標準出力に"_gxpc_explore hongo000" と書き込むことで，通常の gxpc e hongo000 が実行されたときのようにノードの獲得を試みる．また，獲得できたかどうかは各プロセスの標準入力から読み出すことができる．

4.3 e コマンド実行中のコマンドの追加投入

e コマンド実行中に獲得したノードにコマンドを投入できるように実装を行った．ノードの獲得と同様にプロセスの標準出力に"_gxpc_e hostname" と書き込むことで，新しく獲得したノードで hostname が実行される．なお，プロセスの出力はそれ以前に実行されたプロセスと同様にデフォルトでは標準出力に書き出される．

5. 実　　験

本実験では，gxpd とアプリケーションプロセスをソフトウェア的に kill することで，擬似的に故障を実現した．

実験では，故障を検知してから再び，故障が起きたのと同じノードにプロセスを立ち上げるのに要する時間を測定した．なお，各プロセスはただ sleep をするだけのプロセスである．実験は各パラメータで 5 回繰り返し行い，その平均値を求めた．

5.1 heartbeat 間隔

故障検知のための heartbeat 間隔を狭めることで，より故障を迅速に検知することが可能になる．一方，heartbeat 間隔を狭めることで，ネットワークや CPU により大きな負荷がかかることになると考えられる．実験は GigabitEther で繋がれた同一 LAN 内の二つのノードで行った．なお，ノード間の ping の応答時間は 0.081 ms であった．

実験結果は**図 3** のようになった． heartbeat 間隔

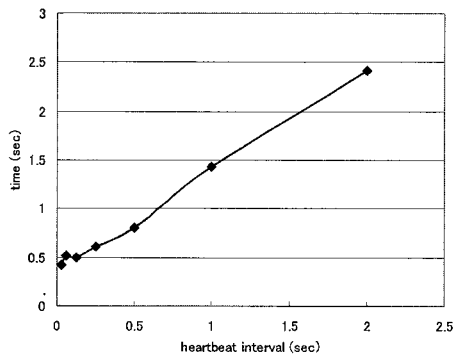

図 3 heartbeat 間隔とプロセスの再配置にかかった時間

に比例して，新しいプロセスが立ち上がるまでの時間が大きくなっている．グラフの線と y 軸との交点が新しいプロセスを立ち上げるのみにかかる時間であると考えられ，約 0.4sec である．改造なしの explore コマンドを用いて，ノードにログインするのに約 0.45sec かかるので，妥当な結果であると考えられる．

このシステムは InTrigger 環境での利用を想定しているため，そのような環境でネットワークや CPU になるべく負荷を与えず動作することが望ましい．例えばノード数 $n = 500$ としてみると，$T_{HB} = 1$ のとき，ルートノードでは，1 秒間に 500 個のメッセージを受け取ることになる．これは，2ms に 1 メッセージのペースである．よって、この故障検知機構は，数秒以下の粒度での故障の検知が必要になるようなアプリケーションでは実用的ではないと考えられる．

5.2 ルートノードからのホップ数による変化

GXP では接続トポロジーが木構造になっている．そのため，ルートノード (gxpc を実行したノード) と通信するのに複数ホップかかるノードでは，故障の検知やプロセスの立ち上げに遅延を生じる可能性がある．

この実験では，ルートノードからのホップ数の違いが，故障検知とプロセスの立ち上げにどの程度の遅延を生じるかを調べる．

以降の実験では，なるべく故障のタイミングと故障検知メッセージ送信のタイミングのずれによる測定のブレをなくし，実装のオーバヘッドを正確に測定するために，heartbeat 間隔を 0.125sec に設定し実験を行った．

5.2.1 クラスタをまたいだ接続

実験環境として，InTrigger プラットフォームのうち 4 つのクラスタの各 1 ノードずつを用いて実験を行った．クラスタ A(@本郷)，B(@本郷)，C(@早稲田)，D(@すずかけ台) に属するノードを 1 台ずつ選びそれぞれ a，b，c，d とする．例えば，ホップ数 3 の実験では a⇒b⇒c⇒d のように 3 段にログインを行い，ノード d を故障させた．結果は**図 4** のようになった． 右

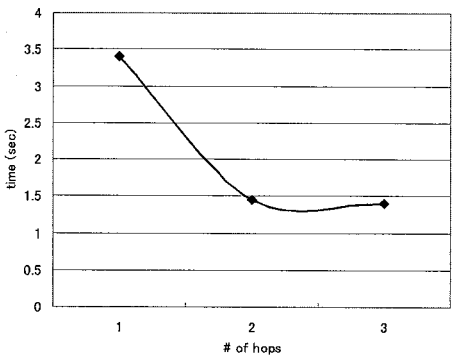

図 4 ルートノードからのホップ数とプロセスの再配置にかかった時間 (WAN をまたいだ接続)

に行くにつれ，ルートノードと故障するノードとの通信のためのホップ数が増加するが，かかる時間との有意な関係は読み取れない．ここで，通常の explore コマンドを用いたときのログインにかかる時間は

a⇒b 3.626sec
b⇒c 1.835sec
c⇒d 1.481sec

であった．この実験環境では，ホップ数による故障検知の遅延の大きさに比べ，ノードへのログインにかかる遅延が非常に大きいと考えられる．そのため，ホップ数の増加による時間の増加がグラフからは読み取れなくなっていると考えられる．

5.2.2 クラスタ内での接続

次に，クラスタ C 内のみで多段に接続することで，実験を行った．結果は**図 5** のようになった．

グラフからはホップ数が増加することで，かかる時

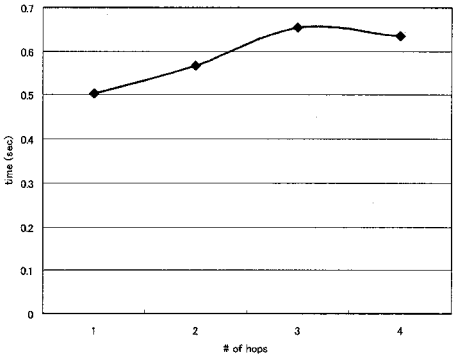

図 5 ルートノードからのホップ数とプロセスの再配置にかかった時間 (LAN 内での接続)

間が大きくなる傾向が読み取れる．ここで，改造なしの explore コマンドを用いてのログインにかかる時間は 0.4sec であった．よって，クラスタ間接続による実験に比べ，ログインにかかる時間が小さく，ホップ数の影響が現れていると考えられる．

以上の二つの実験から，ホップ数の増加によるオーバヘッドはログインにかかる時間に比べて小さく，故障の検知やプロセスの再配置に与える影響は小さいと考えられる．

6. おわりに

本論文では，GXP の紹介と，GXP の e コマンドを用いた長時間計算の実行のための拡張方法を示した．また，実際に実装を行い，故障検知のための heartbeat 間隔と故障を検知するまでの遅延を評価した．さらに，クラスタ内，クラスタ間で多段にログインをしている際の故障検知とプロセスの再配置にかかる時間の評価を行った．

今後の課題としては，以下のようなものがあげられる．

6.1 故障検知機構の性能向上

5 節に示したように，現在の実装では InTrigger プラットフォームの全ノードを用いた規模の計算を想定した場合に，アプリケーションによっては実用に耐えないと考えられる．それは，メッセージがルートノードに集中してしまうためであり，これを解消するような故障検知機構が必要である．それには，例えば中間のノードである程度のメッセージをまとめて 1 つのメッセージにすることで，メッセージ数を減らすなどの方法が考えられる．また，9) などのように各ノードで自立的に故障を検知し，他のノードへ通知できる手法の利用も考えられる．

6.2 ノードの減少への対応

今回の実装では，e コマンド中でのノードの獲得を実装したが，同様にノードの減少への対応も必要になると考えられる．

謝辞 本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新 IT 基盤研究支援プラットフォームの構築」の支援を受けている．

参 考 文 献

1) InTrigger プラットフォーム: `https://www.logos.ic.i.u-tokyo.ac.jp/intrigger/`.
2) Taura, K.: GXP: An Interactive Shell for the Grid Environment, *Proc. IWIA2004*, Vol.1, pp. 59–67 (2004).
3) Butler, R., Gropp, W. and Lusk, E. L.: A Scalable Process-Management Environment for Parallel Programs, pp. 168–175 (2000).
4) Butler, R., Desai, N., Lusk, A. and Lusk, E.: The process management component of a scalable systems software environment, *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 190–198 (2003).
5) Ong, E., Lusk, E. and Gropp, W.: Scalable Unix Commands for Parallel Processors: A High-Performance Implementation, *Euro PVM-MPI*, pp. 17–20 (2001).
6) Saito, H. and Taura, K.: Locality-aware Connection Management and Rank Assignment for Wide-area MPI, *In Proceedings of the 7th International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, pp. 249–256 (2007).
7) 鴨志田良和, 金田憲二, 遠藤敏夫, 田浦健次朗, 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, Vol. 106, No. 199, pp. 19–24 (2006).
8) Ninomiya, T., Tsuruoka, Y., Miyao, Y., Taura, K. and Tsujii, J.: Fast and scalable HPSG parsing, *Traitement automatique des langues (TAL)*, Vol. 46, No. 2 (2006).
9) 堀田勇樹, 田浦健次朗, 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*, Vol.46, No. SIG 12, pp. 236–244 (2005).