東北大学工学部　卒業論文

ウェブインタフェースを介した
スーパコンピュータ利用環境に関する研究

機械知能・航空工学科　滝沢研究室

谷澤悠太

（令和 6 年 3 月）

目次

図目次

表目次

コード目次

第 1 章　緒論

1.1　背景

近年，高性能計算 (High Performance Computing，HPC) システムの用途は多様化し，専門知識を持たない利用者が容易に HPC システムを利用する需要が高まっている．一般的に，コマンド操作に基づいて HPC システムを操作する利用環境や利用する HPC システムごとに異なる操作方法により，HPC を専門としない研究者は HPC システムを使いこなすために多くの学習時間を費やす必要がある．実際に Ping らによると，学習目的で HPC システムを初めて利用する学生などは HPC システム利用環境の構築に多くの時間を費やしてしまい，本来の目的である学問のための HPC システムの利用を達成するまでに多大な時間を費やしてしまうという問題が挙げられている．[1] そこで，従来のコマンド操作に基づく利用環境や，システムごとに異なる利用方法を利用者から隠蔽し，ウェブブラウザを用いて容易かつ統一的に HPC システムを利用することが可能なウェブインタフェースの研究開発が行われている．

しかし，現行のウェブインタフェースはユーザへの簡易かつ統一的な HPC 利用環境を提供するために，ウェブインタフェースの機能の改修を行う度にウェブインタフェース本体を改修する必要がある．そのためシステムの保守性に問題があるといえる．また，HPC システムの利用者にとって，様々なジョブスケジューラを統一的に扱うということは，○○や○○など，様々な分野において有用な研究であり，関心が高い分野である．

そのため，ウェブインタフェースの機能をユーザがウェブブラウザ上で HPC 利用を可能とする機能と多様なジョブスケジューラを統一的に取り扱う機能の二つの機能に分離して実装することで前述した課題点を解決し，ジョブスケジューラ統一化の要望にも応用できる機能の分離利用が可能なウェブインタフェースを考えることができる．

1.2　目的

本研究では，HPC システムの利用難度の高さや HPC システムにおけるジョブスケジューラの多様化に伴い発生し得る HPC システム利用環境に関する課題点に着目する．インタフェースをウェブ機能とスケジューラ抽象化機能に分離することを提案し，操作難易度の高さや保守性などの問題を解決することを目的とする．具体的には現行のウェブインタフェースの機能を分離し，実際のシステムに実装する．実装における動作の確認を行い，提案手法の実現可能性を示す．さらに，提案したウェブインタフェースの機能を定量的に評価することで提案手法の有用性を示す．

1.3　本論文の構成

本論文は全 5 章から構成される．第 1 章では，本研究の背景と目的について述べた．第 2 章では，関連研究について説明し，現行のウェブインタフェースについて述べる．第 3 章では，ウェブインタフェースを介した HPC システム利用環境について説明し，提案手法の実装と動作の確認を行う．第 4 章では，実装した提案手法の評価結果を示し，その考察を行う．第 5 章では，本研究の結論と今後の課題を述べる．

第 2 章　関連研究

2.1　緒言

本章では関連研究のついて述べる．はじめに，一般的な HPC システムの利用方法について説明する．続いて，関連研究である Open OnDemand と呼ばれるウェブインタフェースについて説明する．ウェブインタフェースの機能やその設計について理解し，現行のウェブインタフェースの利点を整理する．最後に，現行のウェブインタフェースにおける課題を述べる．

2.2　HPC システム利用方法

HPC システムとは，スーパコンピュータやコンピュータクラスタの能力を利用して，ほかのコンピュータを遥かに凌ぐ速度で計算課題 (ジョブ) を処理し，実行するシステムを指す．このようなコンピューティング能力の集約によって，さまざまな科学分野において他の方法では対処できない大きな課題を解決できる．実際に，平均的なデスクトップコンピュータは毎秒数十億の計算を実行できる．これは，人間が複雑な計算を行うことができるスピードに比べれば，素晴らしい数字である．しかし，HPC システムは，1 秒に数千兆の計算を実行することができるため，大規模な課題に対してはより適していると言える．

一般的に HPC システムの利用の流れを図 1 に示す．HPC システムは数種類のサーバやデータベースから構成される．システムには，ジョブを実行するために計算を行う計算サーバ (ワーカーノード)，ワーカーノードを管理するためのジョブスケジューラが搭載されたジョブ管理サーバ (マスターノード)，ユーザ情報が保存されたデータベースと連携してログイン情報の管理やユーザリクエストの受け渡しを行うログイン用サーバ，実行するジョブのファイルなどが保存されたファイルサーバなどが存在する．ユーザはログイン用サーバが取り扱うユーザ情報を用いて HPC システムにログインする．ユーザはログインサーバを通じてマスターノードにジョブの実行を依頼する．マスターノードはファイルサーバと連携して，依頼されたジョブのファイル情報などの受け渡しを行い，ワーカーノードにジョブの実行を依頼する

ユーザは利用したい HPC クラスタを遠隔で操作するために自身のコンピュータからユーザ情報を用いて秘密鍵の登録を行った後，ログイン用サーバに SSH 接続をする．そして，ユーザは利用するジョブスケジューラの種類に応じた形式でジョブスクリプトを作成する．その後，与えられたコマンドを用いてジョブを投入を行う．マスターノードのジョブスケジューラが実行するジョブの管理を行い，ワーカーノードはマスターノードの命令に従って

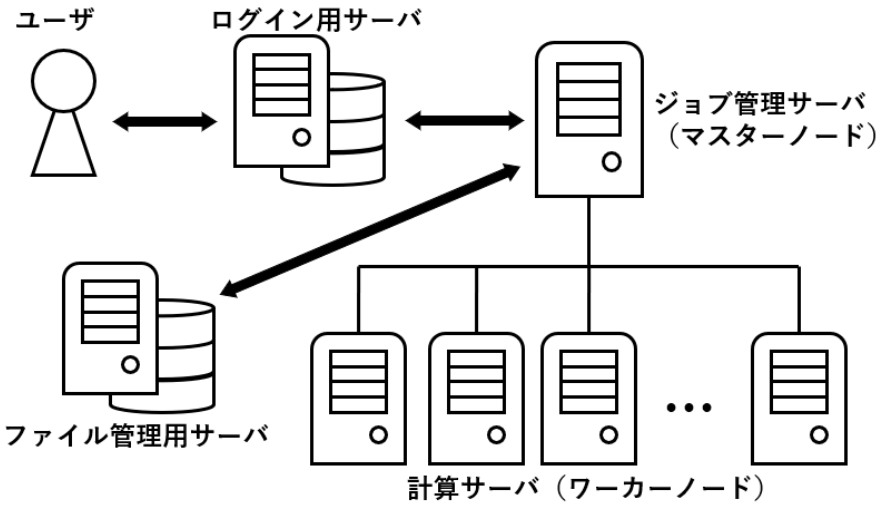

図 1: 一般的な HPC システムの模式図

ジョブの実行を行う．実行が終了したジョブは標準出力とエラーファイルが出力され，ジョブの実行結果を確認することができる．

2.3 Open OnDemand

代表的な関連研究として，Open OnDemand(OOD) とその機能や設計構成を紹介する [2][3]．OOD は米国オハイオ・スーパーコンピューティングセンターが開発したオープンソースソフトウェアであり，ウェブインタフェースを介して HPC システムを利用できる環境を提供する．ユーザはウェブブラウザ上で HPC クラスタを簡単に操作することができ，プラグインやほかのソフトウェアのインストールや設定は不要である．また，リモートデスクトップや jupyternotebook，VSCode などのグラフィカルな対話的操作もウェブブラウザ上から利用することができる．OOD は世界的に使われている様々なジョブスケジューラ (PBS Pro，Slurm，Grid Engine，Torque，LSF など) に対応しているため，システム間の利用方法の差異を隠蔽している．

初めに，OOD の機能について説明する．OOD は，図 2 に示すダッシュボードと図 3 に示すユーザのホームディレクトリを管理する画面，図 4 に示すジョブの状態を確認する画面，図 5 に示すジョブの管理を行う画面，図 6 に示す HPC クラスタのシェル操作を行う画面，開発者が任意の追加アプリケーションソフトウェアを導入することができる interactiveapps から構成される．図 3 に示すホームディレクトリ画面からはユーザのディレクトリを視覚的に操作することができ，ファイルやディレクトリの削除追加編集なども容易に行うことができる．図 4 にはジョブの状態確認を行う Active Jobs 画面を示す．ユーザは投入したジョブの状態をリアルタイムで確認することができる．図 5 の JobComposer の画面では，ユー

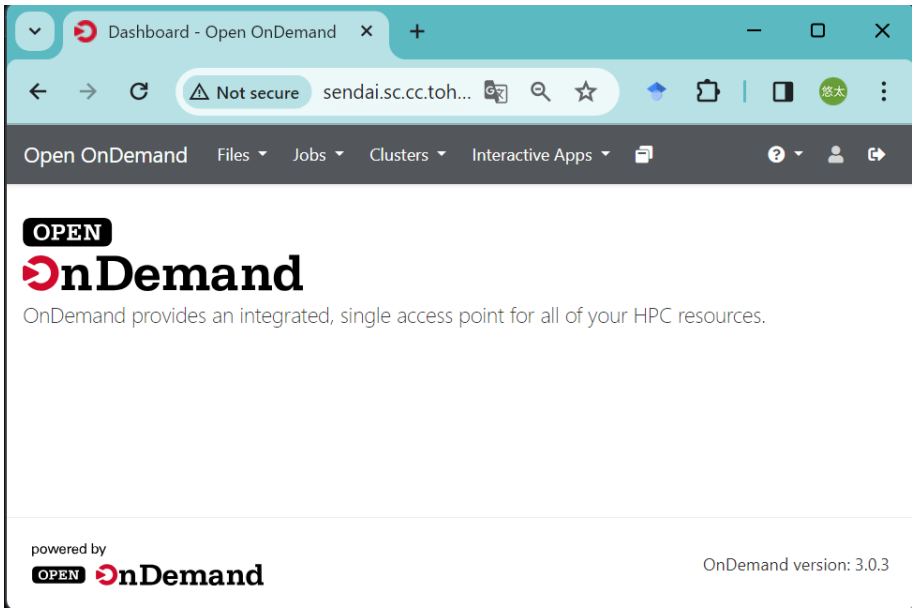

図 2: ダッシュボード画面

ザはジョブの作成，投入，削除などをすべてこの画面から行うことができ，HPC クラスタ内のジョブの管理を行うことができる．図 6 には，シェルの操作画面を示す．ユーザは連携した HPC クラスタのシェルをウェブブラウザ上から操作することができる．このように，OOD は様々な機能やアプリケーションソフトウェアと連携して HPC ユーザの利用者支援を行っている．

続いて，OOD のの設計について説明する．OOD は現在多様なジョブスケジューラに対応しているが，各ジョブスケジューラへの対応は adapters ディレクトリ下に配置される．指定したジョブスケジューラに対応するために，各々で Adapter クラスのサブクラスが宣言され，内部ではジョブスケジューラに合わせてジョブの投入を行うメソッドやジョブの削除を行うメソッド，ジョブの情報を取得するメソッドが再定義されている．対応する Adaoter ファイルを呼び出して参照することで OOD は多様なジョブスケジューラに対応することができている．

以上のように，OOD は視覚的かつ簡単な操作を用いて HPC システムの利用を行うことができるというメリットを持つ．そのため，多くのコンピューティングセンターで実用化され，OOD のユーザ数も年々増加しており，HPC 利用環境を提供するウェブインタフェースとして多くの研究開発が行われている．

2.4　現行のウェブインタフェースにおける課題

現行のウェブインタフェースにおける課題について説明する．OOD は視覚的かつ簡単な操作を用いて HPC システムの利用を行うことができるという大きな利点を持っている．し

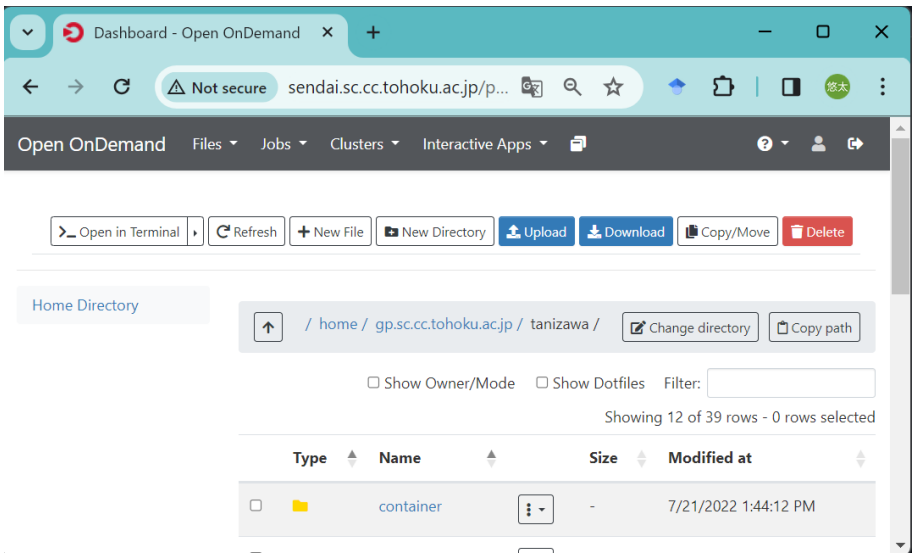

図 3: ホームディレクトリ画面

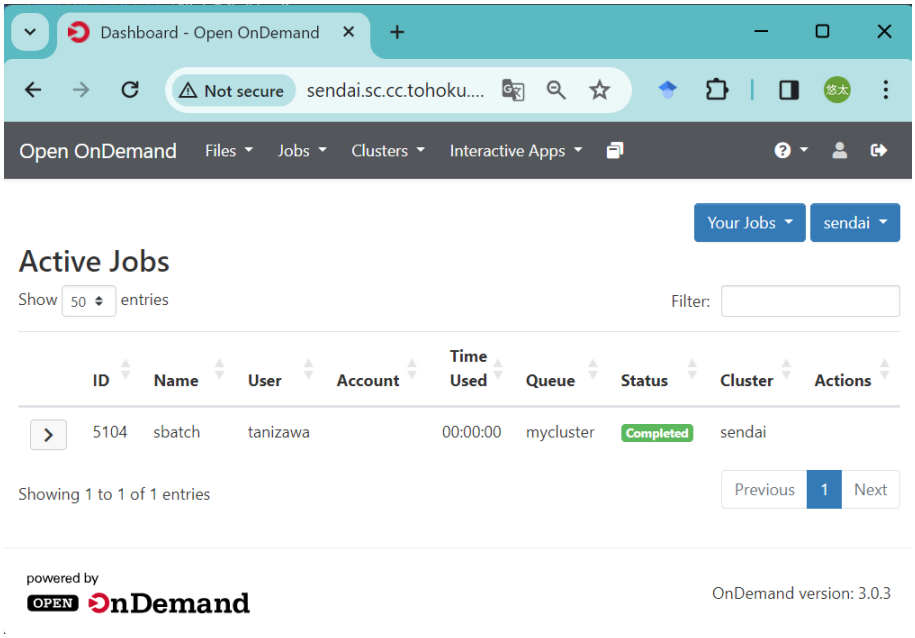

図 4: ジョブ状態確認画面

かし，システム設計上の課題点も考えられる．

国内でのウェブインタフェースの実装事例として，スーパコンピュータ富岳での OOD の実装が挙げられる．OOD は汎用的なツールであるが，富岳で用いられているジョブスケジューラ (Fujitsu Technical Computing Suite, Fujitsu-TCS) に対応していなかったことから，中尾らは OOD を Fujitsu-TCS 向けに改修した事例を報告している [4]．Fujitsu-TCS に対応するために，新たな Adapter ファイルを作成し，Fujitsu-TCS 用のメソッドを再定義することにより OOD は Fujitsu-TCS への対応を行うが，このような改修方法はシステムの基幹部分を直接改修しなければいけないため，慎重に作業を行う必要がある．このよう

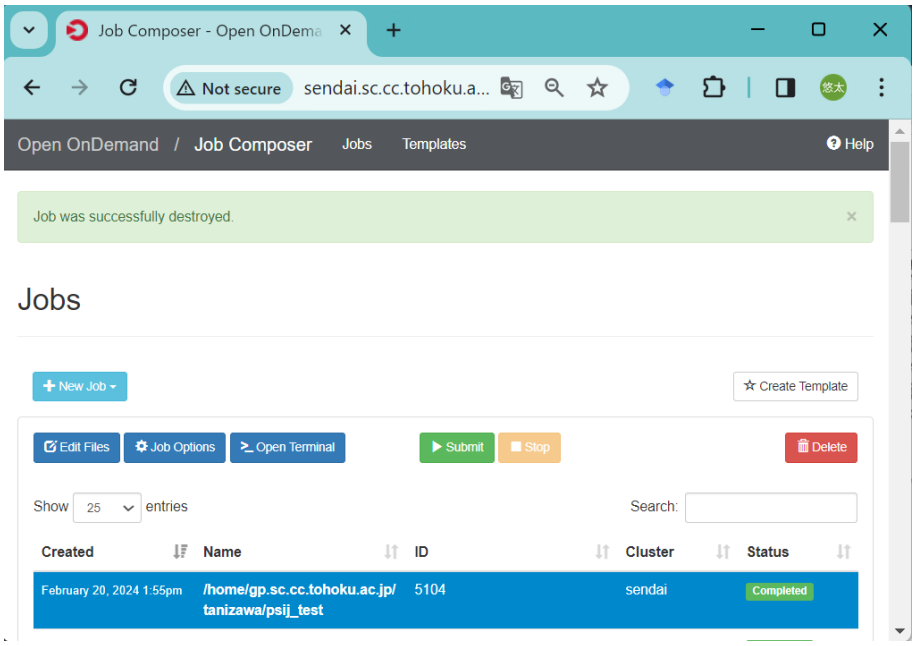

図 5: ジョブ管理画面

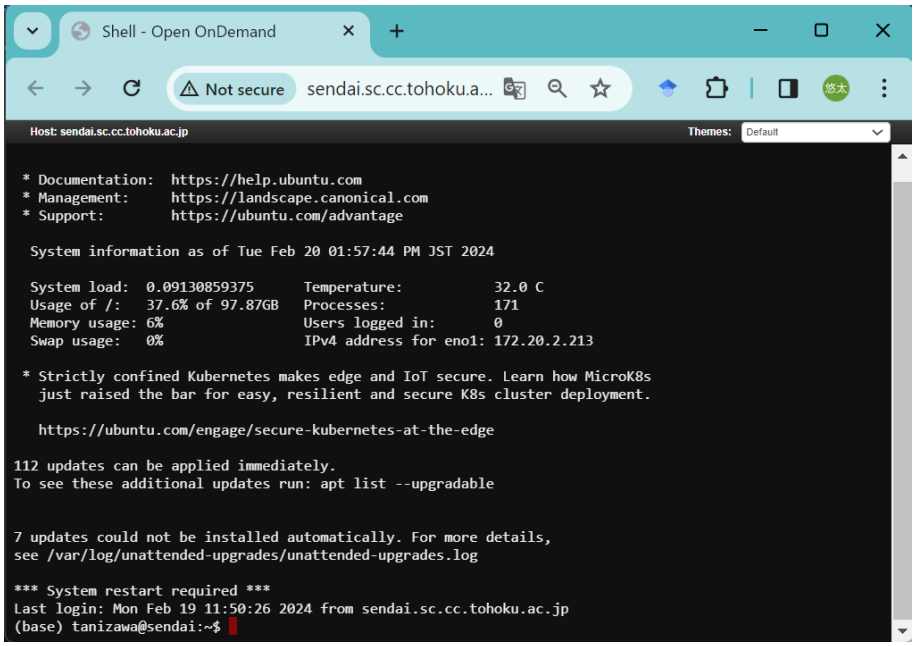

図 6: シェル画面

に，ほかにも様々なジョブスケジューラが存在し，今後も登場することを考えると，ジョブスケジューラがの種類が増えるごとに OOD 本体を直接改修する方法では保守性に問題があるといえる．

2.5 結言

本章では，関連研究について述べた．はじめに一般的な HPC システムの利用方法について説明した．基本的な HPC システムの利用方法について説明し，基礎的な知識を解説した．その後，Open OnDemand と呼ばれるウェブインタフェースについてその機能と設計について説明した．OOD を，用いることによる影響は大きく，HPC システムのユーザに多くの利点をもたらすことができると考えられる．最後に，現行のウェブインタフェースについて，国内での OOD の実装例を参考にして課題点を述べた．ユーザに快適な HPC 利用空間を提供するという利点に反して，システムの保守性が課題点として挙げられる．次章では，本章で述べた現行のウェブインタフェースにおける課題である保守性を考慮した手法を提案し，その実装を行う．

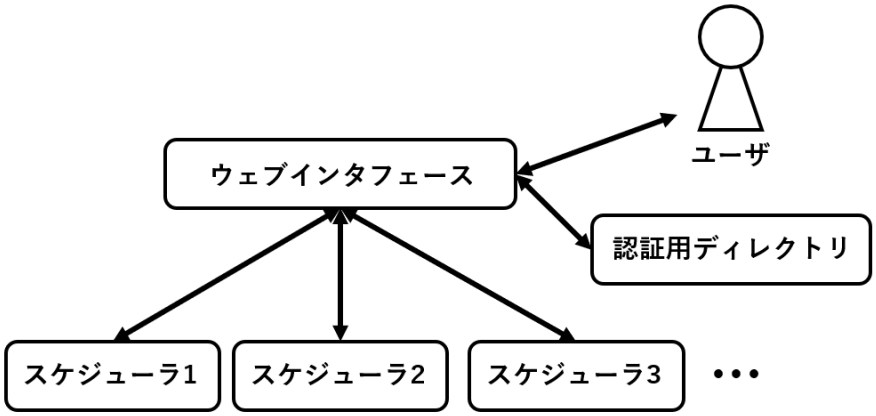

図 7: 従来手法の模式図

第 3 章　ウェブインタフェースを介した HPC システム利用環境

3.1　緒言

前章では，本研究の関連研究について述べた．本章では，ウェブインタフェースを介した HPC システム利用環境の提案手法について説明し，その実装を行う．はじめに，提案手法の概要を説明する．その後，実装の概要，具体的な実装の手順について説明する．

3.2　従来手法

従来のウェブインタフェースを介した HPC システム利用環境について説明する．従来手法の模式図を図 7 に示す．従来のウェブインタフェースは，ユーザとジョブスケジューラのを直接結びつける設計になっている．外部の認証用ディレクトリと連携してユーザ情報を管理することで，ウェブブラウザ上での利用空間の提供を行う．また，主要なジョブスケジューラの抽象化を行うことで，統一的な操作環境を提供する．前章で述べたように，このシステム構成はシステムを改修する際にウェブインタフェース本体を改修する必要があるため，改修中に技術者の予期しない影響を与えてしまう可能性があり，保守性に問題がある．

3.3　提案手法

本研究の目的は，HPC 利用環境をウェブインタフェースに提供する機能 (ウェブ機能) と，ジョブスケジューラ間の差異を抽象化する機能 (スケジューラ抽象化機能) に切り分け，それぞれ独立に保守できる構成を実現することである．このために，本研究ではウェブ機能

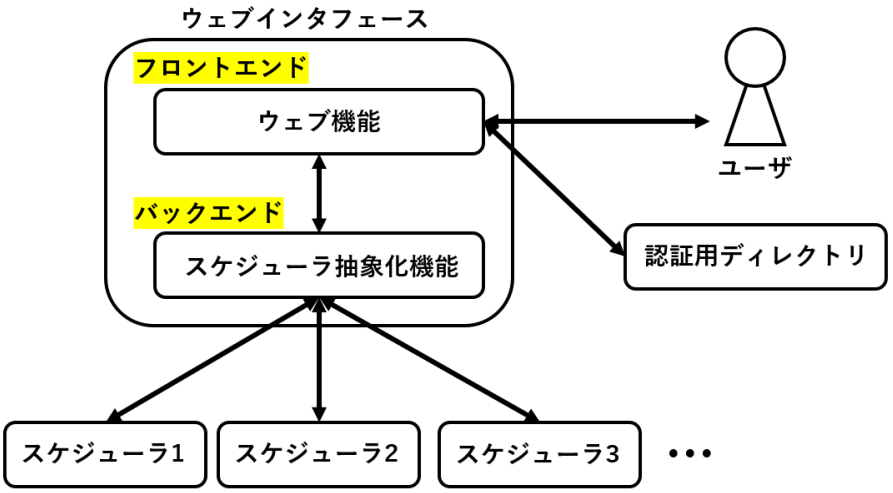

図 8: 提案手法の模式図

からは統一的にシステムを利用し，スケジューラ抽象化機能でシステム間の差異を埋める構成の利用環境を提案する．

この提案手法の模式図を図 8 に示す．フロントエンドでは，ユーザはウェブ機能のみとやり取りを行い，ユーザ情報を管理する外部の認証用ディレクトリを用いて安全に HPC システムを利用することができる．バックエンドでは，ウェブ機能から得られた様々なジョブスケジューラに対するリクエストをスケジューラ抽象化機能が受け取り，処理を行う．この実現のためには，ウェブ機能とスケジューラ抽象化機能を連携させる必要があることから，両者間に求められる情報のやり取りを整理し，適切な連携方法を検討する．

3.4 実装

3.4.1 実装の概要

ウェブ機能とスケジューラ抽象化機能をそれぞれ独立に実装し，連携させることでウェブインタフェースを介して様々なシステムを統一的に利用できる環境を実現する．そのために，ウェブ機能の基盤として OOD，スケジューラ抽象化機能の基盤として PSI/J[5] と呼ばれる Python ライブラリを利用し，両者を組み合わせることで提案手法を実装する．

本研究では，東北大学のスーパーコンピュータ「AOBA」で運用されているジョブスケジューラ (NEC Network QueuingSystem V, NQSV) が OOD に対応していないという事実に着目して，NQSV をスケジューラ抽象化機能側に実装することと，それをウェブ機能側から利用できることを検証する．実装環境として，OOD 用のホストサーバとスーパーコンピュータ AOBA を模した HPC クラスタ (疑似 AOBA クラスタ) を考える．模式図を図 9 に示す．OOD 用のホストサーバでは，OOD の動作が保証されている Ubuntu20.04 LST を OS として用いる．疑似 AOBA クラスタは，マスターノードと二つのワーカーノードか

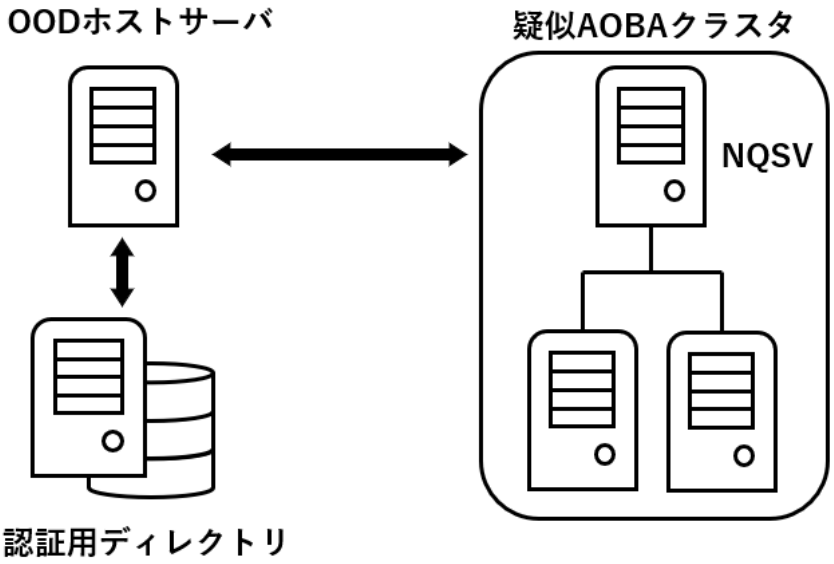

図 9: 実装環境

ら構成される小規模なクラスタであり，AOBA と同様に NQSV がジョブスケジューラとして利用されている．疑似 AOBA クラスタでは NQSV の動作確認が行われている CentOS7 を用いる．OOD はログイン時に認証機構を必要としており，dex との OpeID コネクトや Shibboleth，CAS などの認証方法がある．本研究の実装では，公式が推奨している「Dex との OpenID コネクト」を用いて LDAP 認証を行う [6][7]．認証用ディレクトリである LDAP サーバと連携して認証機構を設計する．

3.4.2 スケジューラ抽象化機能と NQSV の連携

はじめに，スケジューラ抽象化機能と NQSV の連携を考える．スケジューラ抽象化機能の基板である PSI/J は，ジョブの情報を格納する Job クラスとジョブの投入や削除などのメソッドをジョブスケジューラごとに再定義している JobExecutor クラスにより構成されている．本研究では新たに NQSV 用の JobExecutor クラスを作成し，ジョブの投入，削除，ジョブの状態確認を行うための三つのメソッドを実装する．PSI/J が対応している他のジョブスケジューラ (Slurm，PBS Pro，LSF，Flux，Cobalt) はジョブの終了後にジョブの状態 (COMPLETED，CANCELED，FAILED) をコマンドの出力結果から確認できる．しかし，NQSV ではジョブの終了後に COMPLETED，CANCELED，FAILED の状態を確認できない．そのため，NQSV に対応するためには，ジョブの投入，ジョブの削除，および待機中のジョブの存在確認に基づいてジョブの状態を PSI/J 側で把握する必要がある．この機能を実現するため，本研究の実装ではジョブが投入された後にジョブキューからジョブが無くなった際に，そのジョブの状態を COMPLETED に変更する．また，ジョブが削除

された際には，そのジョブの状態を CANCELED に変更する．それ以外にジョブキューの状態をコマンドを用いて定期的に確認し，出力結果に応じて QUEUED あるいは ACTIVE という状態にする．

具体的な実装をコード 1 に示す．JobExecutor クラス内で定義されている get_status_now メソッドは Job クラスのインスタンスを引数にとり，ジョブの状態を保持する JobStatus クラスを返り値に持つ (30 行目)．引数から受け取った Job クラスのインスタンス変数 job.native_id からジョブの ID を抽出し，string 型変数 native_ids に代入する (31 行目)．各ジョブスケジューラごとに指定されているジョブの状態確認コマンドをジョブ ID を指定して投げる．本研究では，NQSV 用のコマンドとして「qstat -F rid,stt -n -l native_ids」を用いて，指定した ID のジョブの ID と状態を返すコマンドオプションを利用する (32 34 行目)．コマンドの出力は行ごとに List 型変数の lines に代入され，中身を for ループでまわす (36 38 行目)．lines 変数の中身は空白区切りの string 型として cols に代入される．指定した ID のジョブがジョブキューにない場合は，ジョブの ID とジョブの状態ではなく「Batch Request: ジョブ ID does not exist on クラスタのホスト名.」という出力が変数 cols に代入されるため，条件文「len(cols)==8」が真であれば指定した ID のジョブがジョブキューにないということがわかる．そのため，条件文が真であり，cancel 時に立つ cancel_frag が立っていればジョブキュー内にジョブがなく，ジョブの削除が正常に行われている状態であるため，ジョブの状態を CANCELED にする (43 51 行目)．一方，条件文が真であり，cancel_frag が立っていない場合は，ジョブキュー内部にジョブがないが，ジョブの削除が行われていないので，ジョブの状態を COMPLETED にする (53 61 行目)．また，ジョブの投入時に立つ submit_frag が立っていない場合はそもそもジョブの投入が成功していないのでジョブの状態を FAILED にする (63 71 行目)．以上の三つの場合分けにおいて，「Batch Request: ジョブ ID does not exist on クラスタのホスト名.」という出力からジョブ ID のみを抽出して，ジョブ ID ごとに list 型変数 r を用意して JobStatus クラスのジョブの状態情報を代入する．また，前述した 3 種類の場合分けのどれにも値しない場合は，ジョブキュー内にジョブが存在する場合であり，ジョブ ID とジョブの状態が変数 cols に代入される．ここで出力されているジョブの状態は NQSV によって定められた状態名でありジョブキューに投入され，実行待ちである状態は QUE，ジョブが実行されている状態は RUN など固有の名称が定められている．これらの固有のジョブ状態名から PSI/J で用いる各ジョブスケジューラで共通の状態名に変換するために，_STATE_MAP と _get_state メソッドを用いる．なお，他のジョブスケジューラではジョブ状態の詳細メッセージが出力される場合があるが，NQSV には詳細メッセージの出力機能がないため message 変数は None としている．

このように PSI/J 自身がジョブの状態を管理することにより，さらに広い範囲のジョブ

スケジューラに対応することができることから，ジョブスケジューラ抽象化機能の汎用性を高めることができたといえる．

コード 1: ジョブの状態取得メソッド

```
1  from pathlib import Path
2  from psij import Job, JobState, JobStatus
3  from typing import List, Collection
4  from psij.executors.batch.batch_scheduler_executor
5      import BatchSchedulerExecutor, check_status_exit_code
6  import re
7  import subprocess
8
9  class NQSVJobExecutor(BatchSchedulerExecutor):
10
11     _STATE_MAP = {
12         'QUE': JobState.QUEUED,
13         'RUN': JobState.ACTIVE,
14         'WAT': JobState.QUEUED,
15         'HLD': JobState.QUEUED,
16         'SUS': JobState.QUEUED,
17         'ARI': JobState.QUEUED,
18         'TRS': JobState.QUEUED,
19         'EXT': JobState.ACTIVE,
20         'PRR': JobState.QUEUED,
21         'POR': JobState.ACTIVE,
22         'MIG': JobState.QUEUED,
23         'STG': JobState.QUEUED,
24     }
25
26     def get_status_command(self, native_ids: Collection[str]) -> List[str]:
27     return ['qstat', '-F', 'rid,stt', '-n', '-l'] + list(native_ids)
28
29     def get_status_now(self, job: Job) -> Job.status:
30         native_ids = ''.join(str(job.native_id))
31         command = ['qstat', '-F', 'rid,stt', '-n', '-l', native_ids]
32         out =
33         subprocess.run(command, capture_output=True, text=True).stdout
34         r = {}
35         lines = iter(out.split('\n'))
36
37         for line in lines:
38             if not line:
39                 continue
40             cols = line.split()
41
42             if(len(cols) == 8 and self.cancel_frag):
43                 s = cols[2]
44                 native_id = ""
```

```
45                  for char in s:
46                      if char.isdigit():
47                          native_id += char
48                  state = JobState.CANCELED
49                  r[native_id] = JobStatus(state=state, message=None)
50                  return r[native_id]
51
52              elif(len(cols) == 8 and not(self.cancel_frag)):
53                  s = cols[2]
54                  native_id = ""
55                  for char in s:
56                      if char.isdigit():
57                          native_id += char
58                  state = JobState.COMPLETED
59                  r[native_id] = JobStatus(state=state, message=None)
60                  return r[native_id]
61
62              elif(not(self.submit_frag)):
63                  s = cols[2]
64                  native_id = ""
65                  for char in s:
66                      if char.isdigit():
67                          native_id += char
68                  state = JobState.FAILED
69                  r[native_id] = JobStatus(state=state, message=None)
70                  return r[native_id]
71
72              else:
73                  assert len(cols) == 2
74                  match = re.search(r'\b(\d+)\b', cols[0])
75                  native_id = match.group(1) if match else None
76                  native_state = cols[1]
77                  state = self._get_state(native_state)
78                  msg = None
79                  r[native_id] = JobStatus(state=state, message=msg)
80                  return r[native_id]
81
82      def _get_messsage(*args, **kwargs):
83      return None
84
85      def _get_state(self, state: str) -> JobState:
86          assert state in NQSVJobExecutor._STATE_MAP
87          return NQSVJobExecutor._STATE_MAP[state]
```

3.4.3 ウェブ機能とスケジューラ抽象化機能との連携

続いて，ウェブ機能側である OOD 側からスケジューラ機能を用いることを考える．実装における問題点として，OOD が Ruby で実装されていることに対して，PSI/J は Python

で実装されているという点が挙げられる [8][9]．そのため，Ruby スクリプト上で Python ライブラリを使用する必要がある．本実装では PSI/J を経由する際のオーバヘッドが小さく，単純な実装であるため，PSI/J を用いたジョブの管理のための Python スクリプトをシェルを経由して Ruby スクリプト上で直接実行する．この実装により，ウェブ機能として OOD を用い，スケジューラ抽象化機能である PSI/J を経由して，指定したジョブスケジューラにジョブの投入や削除を行うことができる．また，PSI/J を仲介することで，OOD が未対応であった NQSV でのジョブ管理を OOD 上から操作することを実現している．

実装をコード 2 に示す．PSI/J と連携するためのアダプタファイルは OodCore/Job/Adapters 空間内で定義され，Adapter スーパークラスのサブクラスとして PSIJ クラスを定義する．図 10 には OOD と PSI/J を接続するための設定ファイル psij.yml の中身を示す．設定ファイルには，OOD からログインを行う際のホスト名，選択する Adapter 名，PSI/J で用いる JobExecutor クラス名，バイナリファイルのパスと用いるジョブスケジューラの設定ファイルのパス，ジョブを投入する HPC クラスタのホスト名を設定する．initialize メソッド内では，設定ファイル psij.yml から与えられた情報をそれぞれインスタンス変数に代入する．submit メソッドでは，引数にジョブスクリプトの中身をとり，返り値に投入したジョブの ID を渡す．与えらえれたジョブスクリプトは一時的なジョブスクリプト保管用のディレクトリに保管され，scp コマンドを用いて，ユーザのホームディレクトリに PSI/J を用いたジョブ投入用の python スクリプト submit_script.py を転送する．その後，ジョブ投入先のホストで JobExecutor と投入するジョブのパスを指定して submit_script.py を実行することでジョブが選択したホストに投入されるようになる．実際に，OOD の Job Composer 画面からジョブの投入を試みる．ジョブを作成する際にクラスタ選択で「psij」を設定することで，PSI/J を経由して Slurm クラスタや NQSV クラスタなど任意の HPC システムにジョブを投入できていることが確認できた．

コード 2: PSI/J と OOD の連携

```
1
2  class PSIJ < Adapter
3    class Batch
4      def initialize(cluster: nil, bin: nil, conf: nil, bin_overrides: {},
5                     submit_host: "", strict_host_checking: true, executor: nil)
6        @cluster              = cluster && cluster.to_s
7        @conf                 = conf    && Pathname.new(conf.to_s)
8        @bin                  = Pathname.new(bin.to_s)
9        @bin_overrides        = bin_overrides
10       @submit_host          = submit_host.to_s
11       @strict_host_checking = strict_host_checking
12       @executor             = executor
13     end
```

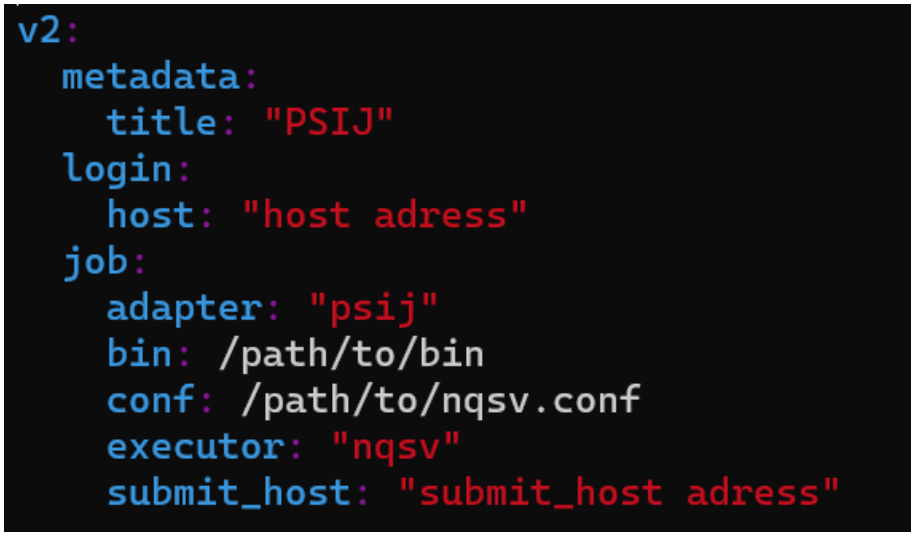

図 10: OOD と PSI/J の接続ファイル

```
14
15    def submit(script, after: [], afterok: [], afternotok: [], afterany: [])
16      job_path = "/Temporary/Path/to/run.sh"
17      file = File.open(job_path, "w")
18      file.puts(script.content.to_s)
19      file.close
20      scp_command = "sshpass -p 'password' scp /Path/to/submit_script.py
21                     username@#{@psij.submit_host}:/Path/to/submit_script.py"
22      system(scp_command)
23      psij_submit =
24      "python3 /Path/to/submit_script.py"
25                              + " " + @psij.executor + " " + job_path
26      ssh_submit =
27      "sshpass -p 'password'
28                     ssh username@#{@psij.submit_host} '#{psij_submit}'"
29      o, e, s = Open3.capture3(ssh_submit)
30      return o
31    end
32
33    def delete(id)
34      ids = id.to_s
35      scp_command = "sshpass -p 'password' scp /Path/to/delete_script.py
36                     username@#{@psij.submit_host}:/Path/to/delete_script.py"
37      system(scp_command)
38      psij_delete =
39      "python3 /ood_tmp/delete_script.py" + " " + @psij.executor + " " + ids
40      ssh_delete =
41      "sshpass -p 'password'
42                     ssh username@#{@psij.submit_host} '#{psij_delete}'"
43      system(ssh_delete)
44    end
```

3.5 結言

本章では，ウェブインタフェースを介した HPC システム利用環境の提案手法について説明し，その実装を行った．はじめに，提案手法の概要を説明した．その後，実装の概要，具体的な実装の手順について説明した．次章では，本章で実装した提案手法の性能評価を行い，提案手法の有用性を考察する．

第 4 章　実装評価

4.1　緒言

本章では，前章で実装した提案手法を用いて，実装評価を行った結果について説明する．はじめに，提案手法の評価を行った際の評価環境と評価条件について説明する．その後，提案手法の実装において考慮すべき実行時のオーバヘッドの測定結果について，表と図を用いて定量的に説明する．

4.2　評価環境

はじめに，評価環境について説明する．ウェブインタフェースをウェブ機能とスケジューラ抽象化機能に分けたことによって両者の連携時のオーバヘッドの発生が懸念されるため，その影響を定量的に評価する．本実装においては，OOD がシェルを介して PSI/J の Python スクリプトを実行するため，そのオーバヘッドによる影響を評価する．

評価には，Selenium[10] と呼ばれるウェブページの自動制御ライブラリを用いる．機能の分離前と分離後を比較したいため，OOD が NQSV に対応していないことから評価環境として疑似 AOBA クラスタを用いることはできない．そこで，実験用に用意した Slurm の HPC クラスタを用いて機能分離に伴うオーバヘッドの評価を行う．ローカルホストから OOD ホストサーバのウェブポータルにアクセスして，ユーザ名とパスワードを用いてログインする．ダッシュボード画面から Job Composer 画面に移動し，NewJob ボタンを押下する．選択肢の中から From Specified Pass を選択してジョブが配置されたディレクトリ，ジョブスクリプトのファイル名，実行するクラスタ名を選択して，ジョブの作成と投入を行う．投入したジョブの状態は squeue コマンドにより確認し，コマンドの出力結果によってジョブの実行完了を確認する．NewJob ボタンを押下した時刻から，そのジョブの実行を完了した時刻までを計測し，ジョブのターンアラウンドタイムとする．

4.3　評価条件

評価条件について説明する．ジョブの投入を 1〜10 回連続で行い，そのターンアラウンドタイムを計測して PSI/J を経由する場合と経由しない場合を比較する．ターンアラウンドタイムは 1〜10 回の連続投入でそれぞれ 20 回ずつ計測し，その平均値をとることで誤差を考慮した結果を得る．また，ジョブを 1〜100 回連続投入した際のターンアラウンドタイムの比較も行った．ジョブの連続投入数は 1 回から開始して 10 回ずつ増やしていき，ジョブ数を大きくした場合の結果を得る．

表 1: PSI/J を経由した場合の評価結果

ジョブ連続投入数	ターンアラウンドタイム [s]
1	24.0
2	28.6
3	51.6
4	64.8
5	78.1
6	91.0
7	105.6
8	118.3
9	131.9
10	146.5

4.4 実行時オーバヘッドの評価

実行時のオーバヘッドの評価を行う．1〜10 回のジョブの連続投入により得られた評価結果の実データを表 1 と表 2 に示す．ジョブを 1 回投入した場合のターンアラウンドタイムはどちらの場合も 30 秒弱であるが PSI/J を経由しない場合の方が若干早いことがわかる．また，ジョブを 10 回連続投入すると，ターンアラウンドタイムは 150 秒弱であり，かなりの時間を要することがわかる．この場合も，PSI/J を経由しない場合の方がターンアラウンドタイムは数秒短いことがわかる．続いてこの実データをグラフとして図 11 および図 12 に可視化する．図 11 では機能分離前後でのターンアラウンドタイムの比較を示す．横軸は連続して投入したジョブの数，縦軸はターンアラウンドタイムを示す．図 12 はジョブ実行時のオーバヘッドを示す．横軸は連続して投入したジョブの数，縦軸はジョブ実行時のオーバヘッドを示す．図 11 から PSI/J を経由した提案手法の方がわずかにターンアラウンドタイムが大きいことがわかり，どちらの場合も連続投入したジョブの数に線形比例して増加していることがわかる．また図 12 から，ジョブの連続投入回数が多くなればなるほど両者の実行時オーバヘッドの差が大きくなっていることがわかる．

ジョブを 1〜100 回連続投入した際の機能分離前後でのターンアラウンドタイムの評価結果を図 13 に示す．横軸は連続して投入したジョブの数，縦軸はターンアラウンドタイムを示す．1〜10 回の連続投入の場合と同じく，PSI/J を経由した場合の方がわずかにターンアラウンドタイムが大きくなっており，連続投入するジョブ数を大きくしても極端にオーバ

表 2: PSI/J を経由しない場合の評価結果

ジョブ連続投入数	ターンアラウンドタイム [s]
1	23.0
2	37.6
3	49.7
4	62.5
5	75.4
6	87.9
7	101.5
8	114.6
9	127.3
10	139.9

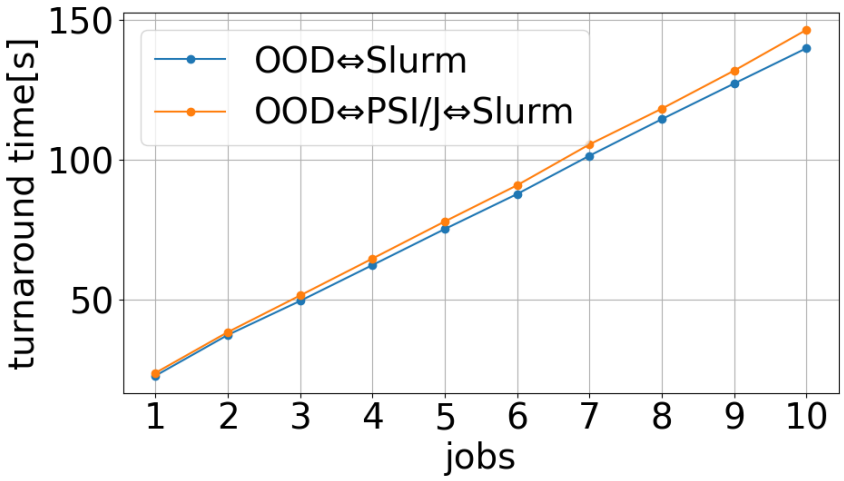

図 11: 機能分離前後でのターンアラウンドタイムの比較

ヘッドに差が出ることはないということがわかった．

様々なジョブの連続投入回数でのターンアラウンドタイムの結果から，ジョブ実行時のオーバヘッドを測定した．その結果から，ジョブの連続投入回数に依らず，PSI/J を経由した場合のオーバヘッドは，PSI/J を経由しない場合のターンアラウンドタイムの 5 %以内に収まり，提案手法によって生じるオーバヘッドは十分無視できるといえる．

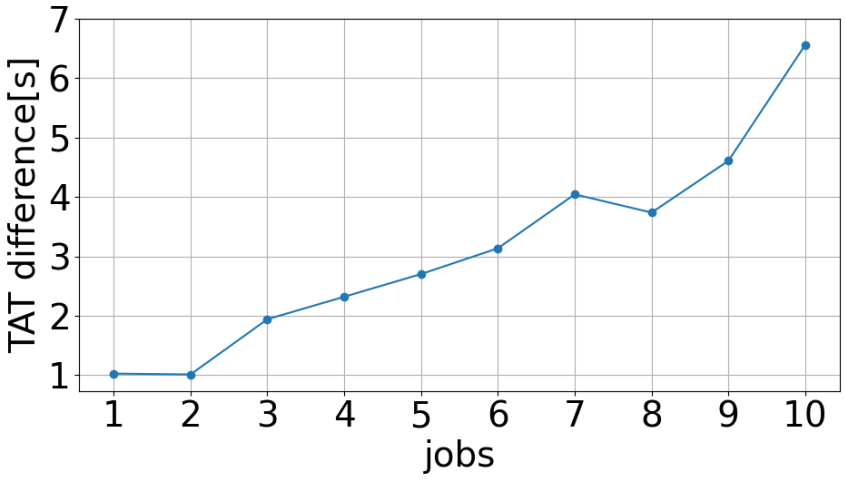

図 12: 実行時オーバーヘッドの差

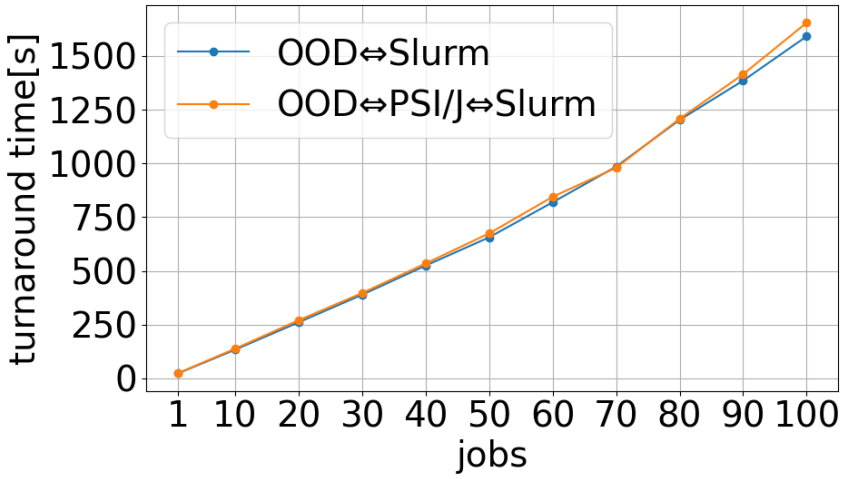

図 13: ジョブ数を増加した際のターンアラウンドタイムの比較

4.5 結言

本章では，実際にジョブの投入を行った際の提案手法の実装の評価結果を示した．はじめに，評価を行う環境について説明した．その後，機能の分離前後で生じるオーバヘッドを測定する際の評価条件について説明した．これらの評価環境と評価条件により得られた評価結果により，提案手法により生じるオーバヘッドは充分小さいため実用上は問題ないということを示した．

第 5 章　結論

近年，HPC システムの用途は多様化し，専門知識を持たない利用者が HPC システムを利用する需要が高まっている．HPC の利用にはコマンド操作に基づいた利用環境や，システムごとに異なる操作方法など HPC システムを利用するためには多くの学習時間が必要とされている．このような課題点を解決するためにウェブブラウザを用いて容易かつ統一的に HPC システムを利用することが可能なウェブインタフェースについての研究開発が多くお根割れている．しかし，現行のインタフェースは二アブ機能の改修を行うたびにインタフェース本体を改修する必要があるため，保守性に大きな問題を抱えている．

本研究では，HPC システムの利用難易度の高さ，スケジューラ多様化に伴うシステムの保守性の問題という点に着目し，HPC システムの敢為な利用環境と充分なシステムの保守性を併せ持つウェブインタフェースの実装について考えた．この実現のために，ウェブ機能とスケジューラ抽象化機能を分離し，それぞれ独立に保守管理する手法を提案した．

第 2 章では，本研究の関連研究について説明した．はじめに，HPC システム利用環境について説明した．その後，ウェブインタフェースである OOD について説明し，OOD の機能と設計について説明した．さらに，現行のインタフェースの課題点について説明し，本研究で着目すべき点を明らかにした．

第 3 章では，ウェブインタフェースを介した HPC システム利用環境として提案手法の説明と実装を行った．はじめに，従来手法と提案手法の説明を行った．従来手法と提案手法を比較して，提案手法の利点や設計の変更点について説明した．その後，提案手法の実装を行った．はじめに，実装の概要について説明した．その後，スケジューラ抽象化機能と NQSV の連携を行った．スケジューラ抽象化機能自身がジョブの状態を管理する設計を行ったことにより，スケジューラ抽象化機能の汎用性をより高めることができたといえる．続いて，ウェブ機能とスケジューラ抽象化機能との連携を行った．ウェブ機能がスケジューラ抽象化機能を呼び出すことで，ウェブ機能として OOD を用い，スケジューラ抽象化機能である PSI/J を経由して，指定したスケジューラにジョブの投入や削除を行うことができるようになった．

第 4 章では，機能分離に伴いオーバヘッドの発生が懸念されるため，実装の評価を行った．はじめに，評価環境，評価条件について説明した．その後，ジョブ実行時のオーバヘッドの評価を行った．提案手法で発生が懸念されたオーバヘッドは機能分離前のターンアラウンドタイムの約 5 %以下であることがわかり，提案手法の実装によって生じたオーバヘッドは充分に小さいということが明らかになった．

以上の結果から，HPC システム利用環境におけるウェブインタフェースをウェブ機能と

スケジューラ抽象化機能の二つに分離し，それぞれ独立に保守管理することの実現可能性と有用性をしめすことができた．

今後の課題として，PSI/J へのジョブの一旦停止 (hold) 機能と再開 (release) 機能の実装が挙げられる．OOD では hold 機能と release 機能が実装されていることに対して，PSI/J ではそれらの機能に対応していない．そのため，これらの操作に必要な両者の連携を設計して実装していくことで，より多様なジョブスケジューラやその使い方に対応可能となると期待される．

参考文献

[1] Ping Luo, Benjamin Evans, Tyler Trafford, Kaylea Nelson, Thomas J. Langford, Jay Kubeck, and Andrew Sherman. Using Single Sign-On Authentication with Multiple Open OnDemand Accounts: A Solution for HPC Hosted Courses. *IEICE TRANS. INF. SYST*, No. 9, pp. 2307–2314, 9 2018.

[2] David E. Hudak, Thomas Bitterman, Patricia Carey, Douglas Johnson, Eric Franz, Shaun Brady, and Piyush Diwan. OSC OnDemand: A Web Platform Integrating Access to HPC Systems, Web and VNC Applications. *XSEDE '13,*, No. 49, pp. 1–6, 7 2013.

[3] Robert Settlage, Eric Franz, Doug Johnson, Steve Gallo, Edgar Moore, and David Hudak. Open OnDemand: HPC for Everyone. *ISC 2019 Workshops*, Vol. 11887, pp. 504–513, 12 2019.

[4] Masahiro Nakao, Masaru Nagaku, Shinichi Miura, Hidetomo Kaneyama, Ikki Fujiwara, Keiji Yamamoto, and Atsuko Takefusa. Introducing Open OnDemand to Supercomputer Fugaku. *SC-W 2023*, pp. 720–727, 12 2023.

[5] Mihael Hategan-Marandiuc, Andre Merzky, Nicholson Collier, Ketan Maheshwari, Jonathan Ozik, Matteo Turilli, Andreas Wilke, Justin M. Wozniak, Kyle Chard, Ian Foster, Rafael Ferreira da Silva, Shantenu Jha, and Daniel Laney. PSI/J: A Portable Interface for Submitting, Monitoring, and Managing Jobs. *IEEE 19th International Conference on e-Science*, 2023.

[6] OpenOnDemand 3.0.3 Docuentation. `https://osc.github.io/ood-documentation/latest/`.

[7] install Open OnDemand — Open OnDemand. `https://openondemand.org/install-open-ondemand`.

[8] OSC/ood-core: Open OnDemand core library. `https://github.com/OSC/ood_core`.

[9] ExaWorks/psij-python: psij-python library. `https://github.com/ExaWorks/psij-python`.

[10] S. Nyamathulla, Dr. P. Ratnababu, Nazma Sultana Shaik, and Bhagya Lakshmi. N. A Review on Selenium Web Driver with Python. *Annals of the Romanian Society for Cell Biology*, Vol. 25, pp. 16760–16768, 6 2021.

謝辞