Vol.2014-HPC-144 No.14
2014/5/27

HPCS Toolkit によるソースコード自動書換に向けた ファイル I/O 性能最適化方式の検討

根岸康[†1]　村田浩樹[†1]
Guojing Cong[†2]　Chung I-Hsin[†2]　Wen Hui-Fang[†2]

プログラムのファイル I/O 性能を性能最適化ツール HPCS Toolkit による自動ソースコード書換えにより向上させることを目指す．特にファイル入力時の blocking read の non-blocking read への変換により複数の I/O 要求を並列に発行し，blocking read の待ち時間を削減し，ファイル I/O 性能を向上させる． 本稿でこの最適化を HPCS ツールキットにルールとして登録する際に必要となる各機能について検討した．ファイル I/O ベンチマークプログラム及び正規表現検索プログラム(agrep)を例にとって，必要なプログラム書換えの量やその性能最適化の効果について評価する．提案する最適化により正規表現検索プログラムの最適化では，read 時間が最大 52%，全体の実行時間が最大 22%向上した．

File I/O performance improvement mechanism for source code optimization by HPCS toolkit

YASUSHI NEGISHI[†1] HIROKI MURATA[†1]
GUOJING CONG[†2] CHUNG I-HSIN[†2] WEN FUI-FANG[†2]

We optimize the file I/O performance with automatic source code optimization by HPCS Toolkit. In this paper, we improve the file I/O performance with conversion from blocking read to non-blocking read by issuing multiple I/O requests at the same time and reducing the waiting time for blocking read. The performance of the agrep program was improved by maximum 52% of read time, and 22% of the whole execution time.

1. はじめに

ビッグデータ等データ解析需要の高まりやシステム中のプロセッサ数の増加等により，スーパーコンピュータ等の大規模システムにおけるファイル I/O 性能及びその最適化の重要性が高まっている．システム構成やアプリケーションは年々複雑化しており，その最適化のために深い知識を持ったプログラマの時間がより多く必要となっている．また，プログラミング環境という観点から見ると，CPU やメモリアクセスの最適化はコンパイラが担当するが，ファイル I/O 最適化はコンパイラにより十分にカバーされているとは言えない．これまで，ファイル入出力の性能を向上させようとする研究3)4)5)6)7)8)はいくつかあったが，ソースコードを自動変換して性能を向上させようとする研究はなかった．

性能最適化の生産性向上のために，性能専門家の知識，コンパイラ技術，性能解析とその最適化方式発見のための研究成果を統合した，HPCS toolkit1)2)と呼ばれるフレームワークがある．このフレームワークを使って開発された性能解析及び最適化方式は，ルールという形で拡張可能なデータベースに格納される．

本研究ではプログラムのファイル I/O 性能を性能最適化ツール HPCS Toolkit による自動ソースコード書換えにより向上させることを目指して，ボトルネックの検知，最適化後の性能の推定，ソースコードの書換え方式等ツールへの登録に必要な各機能についてどのように実現すればよいかを検討する．特にファイルの blocking read を non-blocking read に変換することにより複数の read 要求を並列に発行し，blocking read の待ち時間を削減し，ファイル I/O 性能を向上させる最適化について検討する．また，最適化による性能向上効果を正規表現検索プログラムである agrep によって評価し，更にその際のソースコード変換の方法についても評価する．agrep プログラムの最適化では，read 時間が最大 52%，全体の実行時間が最大 22%向上した．

以下，第 2 章でファイル I/O の自動性能最適化のための方針について，第 3 章で具体的なプログラム変換の方式について，第 4 章ではファイル I/O ベンチマークプログラム及び agrep プログラムによる性能評価について，第 5 章ではまとめと今後の課題について説明する．

2. ファイル I/O 性能自動最適化の方針

本章では，ファイル I/O 性能の自動最適化のための方針について説明する．

2.1 HPCS Toolkit

性能最適化の生産性向上のために，性能専門家の知識，コンパイラ技術，性能解析とその最適化方式発見のための研究成果を統合した，HPCS toolkit と呼ばれるフレームワークがある．このフレームワークを使って開発された性能解析及び最適化方式は，拡張可能なデータベースに格納され

†1 日本アイ・ビー・エム(株)東京基礎研究所
IBM Research - Tokyo
†2 IBM T.J. Watson Research Center

る（図 1　HPCS Toolkit 参照）.

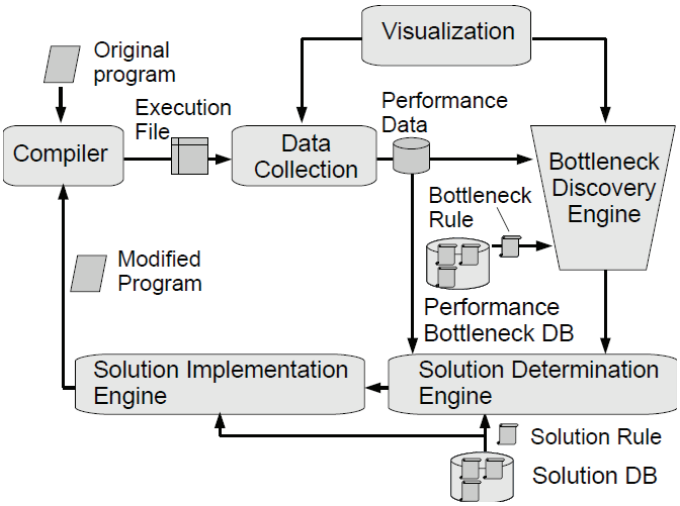


図 1　HPCS Toolkit

Figure 1　HPCS Toolkit.

このフレームワークはプロセッサの L1/L2 キャッシュミス，通信待ち等様々な性能ボトルネックの解析・改善を目的としているが，本研究ではこのフレームワーク上に，ファイル I/O 性能自動最適化のための性能解析および最適化方式をルールという形でデータベースに登録することを目指す．HPCS Toolkit では，ボトルネックの発見，解析に用いる機能は Performance Bottleneck DB に Bottleneck Rule という形で保存され，プログラムの最適化に用いる機能は Solution DB に Solution Rule として登録される．本稿では，その際に必要となる機能の実装方法について検討した．ただし，ルールとしての登録方式等 HPCS Toolkit に固有の実装については本稿では最小限の説明に留めるので，別途参考文献 1)2)を参照のこと．性能解析および最適化の手法が HPCS Toolkit にルールとして登録されれば，他の利用者も対象プログラムをこの Toolkit で性能解析することにより，ファイル I/O によるボトルネック箇所を発見できる．また，可能であれば自動ソースコード書換えによりプログラムを最適化することができる．

2.2 I/O オーバーラップによる最適化

本研究では blocking read を non-blocking read に変更することで CPU のアイドル時間や他の read 処理と処理時間をオーバーラップさせることにより全体の実行時間を削減することを目指す（図 2 参照）．なお，今回はファイル入出力処理に注目してその実行時間を削減することを目指すので，計算時間の並列化は行わない．

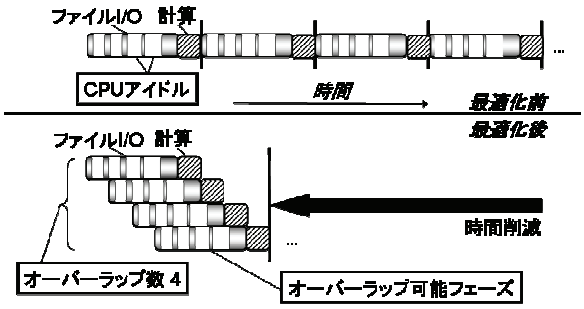


図 2　ファイル I/O のオーバーラップ

Figure 2　File I/O Overlapping.

2.3 プログラム例

以下，本最適化による Fortran プログラムでのソースコード変換例を挙げる．ただし，この例は変換前後のプログラムの動作が分かりやすくなるように記述したものであり，実際の適用に際しては変更箇所や量を少なくするための工夫が必要となる．ソースコード変換の方法については2.8 節で C 言語について説明する．また，このプログラムは3 章でもマイクロベンチマークとして，性能評価に用いる．

```
real*8 val(recnum)
…
do i = 1, size
  open(2, file=TRIM(files(i)), …
  read(2, rec=1) (val(j), j=1, recnum)
  close(2)
  do j = 1, recnum
    total = total + val(j)
  enddo
enddo
…
```

図 3　オリジナルプログラム(擬似コード)

Figure 3　Original Program.

```
integer aio_winnum
parameter(aio_winnum = 15)
integer aio_id(aio_winnum), aio_rpos,
-     aio_rnum, aio_r, aio_win
real*8 aio_val(recnum, aio_winnum)
…
aio_rpos = 1
do i = 1, filenum
  aio_rnum = 1
  if (i .eq. 1) aio_rnum = aio_winnum
  if (i .gt. filenum-aio_winnum+1) aio_rnum=0
  do aio_r = 1, aio_rnum
    aio_win = mod(aio_rpos, aio_winnum) + 1
    fd = aio_win + 6
    open(fd, file=TRIM(files(aio_rpos)), … ,
         asynchronous='yes')
    read(fd, rec=1, asynchronous='yes'
```

```
-        id=aio_id(aio_win))
-        (aio_val(j, aio_win), j=1, recnum)
     aio_rpos = aio_rpos + 1
   enddo
   aio_win = mod(i, aio_winnum) + 1
   fd = aio_win + 6
   wait(fd, id=aio_id(aio_win))
   close(fd)
   do j = 1, recnum
     total = total + aio_val(j, aio_win)
   enddo
 enddo
...
```

図 4　最適化プログラム(擬似コード)

Figure 4　Optimized Program.

この書き換え例では，aio_win 個の環状バッファを用意し，最大 aio_win 個のデータの non-blocking read を先行して実行する．その後，対応する non-blocking read の完了を順次 wait し計算を進めるようにプログラムを変換している．
この書換えにより最大 aio_win 個の non-blocking read が計算と並行して実行されることとなる．

2.4 最適化の手順

ソースコード変換による最適化の手順を HPCS Toolkit のルールとして登録するためには，(1) 最適化対象となるボトルネックの発見，(2) ソースコード変換の妥当性の検証，(3)最適化効果の推定値の提示, (4)ソースコード変換の適用という 4 つの手順を用意する必要がある．以下，この各手順についてその方法を検討する．

2.5 最適化対象となるボトルネックの発見

HPCS Toolkit では，ボトルネック発見ルールを式の性能測定値を変数とする式の形で与える．今回使用する式は以下の通りである．

(ElapsedTime - PM CYC/CPUFrequency) / ElapsedTime > α
and #blocking reads > 0:

式の中の各値は，対応するホットスポットについて，*ElapsedTime* は実行の実時間，*PMCYC* は実行クロック数，*CPUFrequency* はプロセッサの周波数を，また，*#blocking reads* は blocking read の個数を示す．“*(ElapsedTime - PMCYC/CPUFrequency) / ElapsedTime > α*”で CPU のアイドル時間の割合が一定以上であること確認し，“*#blocking reads > 0*”でホットスポット中にブロッキングリードが存在することを確認する．

2.6 ソースコード変換の妥当性の確認

HPCS Toolkit ではソースコード変換の妥当性を確認するための C の関数をルールとして登録する．ソースコード変換の妥当性について確認すべき項目については，現在検討中であるが，ソースコード変換方式に強く依存する．特に，非同期 read の際に使用する中間バッファにアプリケーションが直接アクセスするのか，中間バッファからアプリケーションの入出力バッファにコピーするのかによって大きく異なる．
アプリケーションの入出力バッファにコピーする場合は，read 文のセマンティクスを維持できるため，必要な確認項目は同じファイルに対する入出力の有無のみとなる．一方，アプリケーションに中間バッファに直接アクセスさせる場合には，ファイル入出力文中の引数の前方及び後方への依存関係を調査し，イテレーションを跨いだ依存関係が無いことを確認する必要がある．

2.7 最適化効果の推定値の提示

HPCS Toolkit では，可能な場合適用可能な最適化について得られる性能向上の推定値を提示する C の関数をルールとして登録する．
本最適化では計算の並列化は行われないので，計算時間は最適化の前後で変化しない．また，並列度の増加に伴うオーバーヘッドの増加はないものと仮定した．図 2 で<並列度>回のフェーズが 1 つに重ねられたと考え，また，各フェーズの待ち時間の合計は単純に並列度で分割されると考える．計算は並列化されないので計算時間は測定した待ち時間 * (<並列度> - 1)となる．

<推定待ち時間> = [測定待ち時間] / <並列度>
<推定計算時間> = [測定計算時間] * (<並列度> - 1)
<推定応答時間> = <推定待ち時間> - <推定計算時間>
(ただし，<推定応答時間>が負の場合 0 とする)
<推定最良並列度> = INT([測定待ち時間]/[測定計算時間])
本推定式の妥当性については，3 章で評価する．

2.8 ソースコード変換の適用

HPCS Toolkit ではソースコード変換を行うための C の関数をルールとして提供する．その際，ソースコードの解析木にアクセスするため及びソースコードを変換するためのライブラリが提供される．現在ターゲットプログラムとしては C 言語についてソースコード変換の具体的な方法を検討しており, Fortran については C 言語での検討が終了後に検討する予定である．このため本節では，C 言語を例にとってソースコード変換の方法について検討する．
C 言語の場合も2.3 節の Fortran の例と同様にソースコードの変更のみで最適化を実施することを検討した．しかし，最適化を全てソースコード変換で行う場合，書換え箇所や量が大きくなること，また，変換の妥当性の検証が複雑になることが判明した．このため，今回はファイル入出力を行うライブラリを用意し，ソースコード中の open，read，close 等のファイル入出力文とほぼ等価な API をライブラリとして, 用意しそれぞれ置換する方式について検討した．この方法により，ソースコード変換の箇所や大きさを削減するとともに，ソースコード変換の妥当性の確認もより容易になる．

2.9 非同期 read 用ライブラリ

最適化の際は複数のファイルを同時にオープンするため，複数のファイルディスクリプタや環状バッファを用意する必要があるので，このライブラリがファイルディスクリプタや環状バッファをライブラリ内の変数として確保する．Read したデータは一旦環状バッファに読み込まれた後で，アプリケーションのバッファにコピーする．この方法では環状バッファからアプリケーションのバッファへのコピーが必要となるが，非同期 read の動作がライブラリの外部には隠蔽されるため，アプリケーション中の read 文をそのまま置換することが可能になり，ソースコードの変更箇所や量を削減でき，ソースコード変換の妥当性の確認も容易になる．hpcsaio_hint もしくは hpcsaio_hint_one API の使用は必須ではない．これらの API のいずれかを用いて，open 処理に必要な情報を事前に与えられた場合は，一つのファイルの read 処理の完了を待たずに，複数のファイルに跨った非同期 read 処理を用いたオーバーラップ動作が可能になる．この場合の性能についても3 章で説明する．このライブラリは以下の API を持つ．

(1) int hpcstaio_init(size_t winsize, int winnum);
初期化処理関数．Winsize で読み込みバッファ 1 個の大きさを，winnum でバッファの個数を指定する．
この関数で必要なバッファを確保する．

(2) int hpcstaio_term();
終了処理関数．
この関数でバッファを開放する．

(3) int hpcstaio_hint(int filenum, char *filename[], int flag, mode_t mode);
後述のマルチファイルモードを使用する場合に使用する．オープンするファイルを配列で指定する．

(4) int hpcstaio_hint_one(char *filename, int flag, mode_t mode);
マルチファイルモードを使用する場合に使用する．オープンするファイルを 1 つで指定し，追加する．

(5) int hpcstaio_open(char *path, int flag, ...);
open 関数をこの関数で置換する．なお，戻り値のファイルディスクリプタは便宜上のものなので，本ライブラリ内の関数のみで使用可能となる．select 等他の API では使用できない．
この関数はファイルを跨いだ先読みをしない場合は通常のオープンと同じ動作を行うが，ファイルを跨いだ複数ファイルの先読みをする場合は，この関数の最初の実行時に並列度数のファイルをオープンし各ファイルに対して事前に指定された大きさの非同期 read を発行する．二回目以降の実行時は，順次次のファイルをオープンし非同期 read を発行する．

(6) ssize_t hpcstaio_read(int fd, void *buf, size_t size);
read 関数をこの関数で置換する．
この read 関数の実行時に，発行済みの非同期 read が引数 size で指定された量を環状バッファに読み込むまで待ち合わせし，環状バッファに読み込まれた内容をアプリケーションのバッファ buf にコピーする．環状バッファに空きができた分，次の非同期 read を発行する．

(7) int hpcstaio_close(int fd);
close 関数をこの関数で置換する．
指定されたファイルディスクリプタのファイルを close し，関連するバッファを開放する．

3. 性能評価と考察

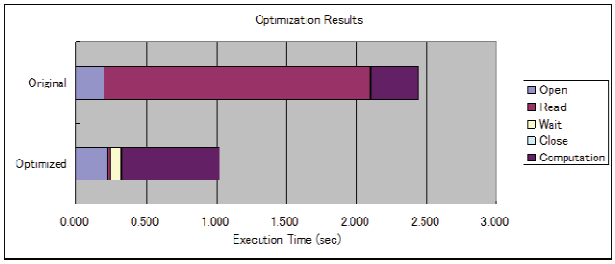
2.3 節で説明したプログラム例及び正規表現検索プログラム agrep9)について，ソースコードを手動で変換してその前後の性能を測定した．本節ではその測定結果及び性能その他に関する考察について説明する．

3.1 実験環境

プロセッサ:Power5+(1.9GHz)
OS: AIX
ネットワーク:100Mbps Ethernet
ファイルシステム: JFS もしくは NFS
コンパイラ:xlc
オプション:-O3 –qhot
測定方法: 10 回測定し平均値を使用

3.2 マイクロベンチマークによる評価

2.3節で示したプログラムをマイクロベンチマークとして，その最適化前後の実行時間を測定し，最適化の効果を評価した．上記実験環で 4MB のファイルを 512 個使用，並列度 5 の時，ファイル入出力時間が 84%，全体の実行時間が 48%削減された．



	Open	Read	Wait	Close	Computation	Total
オリジナル	0.195	1.906	0.000	0.008	0.331	2.440
最適化後	0.222	0.022	0.076	0.008	0.694	1.022

図 5 マイクロベンチマークの最適化前後の実行時間(秒)
Figure 5 Execution Time of Micro Benchmark before and after the Optimization.

実行時間の内訳を見ると，Read 時間のほとんどが削減され，Computation 時間と Wait 時間が若干延びている．

3.2.1 最良並列度について

ファイルシステムとして Journal File System (JFS)を使用し，一回の read で読み込む量を 1KB から 2048KB まで変化させ

た場合の実行時間，最良並列度，推定最良並列度，実際の最良並列度に基づく推定最小実行時間，推定最良並列度にもとづく推定最小実行時間は，以下のとおり．

File List	Read Size (KB)	Original Program	Best Overlapping Number	Estimated Best Overlapping Number	Best Optimized Time	Time Optimized by Estimation	Ratio between Best and Estimated
JFS 2M	1	0.0270	Original	1	–	–	0.00%
JFS 2M	2	0.0263	Original	1	–	–	0.00%
JFS 2M	4	0.0278	Original	1	–	–	0.00%
JFS 2M	8	0.0284	Original	1	–	–	0.00%
JFS 2M	16	0.0329	Original	1	–	–	0.00%
JFS 2M	32	0.0313	Original	1	–	–	0.00%
JFS 2M	64	0.0344	Original	2	–	0.0345	-41.15%
JFS 2M	128	0.0461	Original	2		0.0550	19.25%
JFS 2M	256	0.0657	3	3	0.0510	0.0510	0.00%
JFS 2M	512	0.1059	2	3	0.0771	0.0760	-2.56%
JFS 2M	1024	0.1917	3	3	0.1201	0.1201	0.00%
JFS 2M	2048	0.3659	3	3	0.2154	0.2154	0.00%

図 6　マイクロベンチマークの推定最小実行時間

Figure 6　Estimated Execution Time of Micro Benchmark

表中，赤色はオリジナルの方が性能がよい場合を，黄色は推定最良並列度が最適でない場合を，緑色は最適な推定最良並列度が選択された場合を示している．グラフから分かるように推定最良並列度はほとんどの場合に最良並列度と一致している．また，違っていた場合でもほとんどの場合，得られる最適化の効果はほぼ同等である．問題は一回の Read が 64KB と 128KB の場合で，本来最適化を行わないことが最良であるにも関わらず，推定最良並列度は 2 となっており並列度 2 として最適化を行うと最適化後の性能が元のプログラムより 19-41%悪くなる．これらの点については今後改善が必要となる．

3.3 正規表現探索プログラム agrep による評価

本節ではファイル中の正規表現を探索する C 言語で記述されたプログラム agrep9)に関して，提案する最適化による性能向上や必要なソースコード変換について評価した．2.8 節で示した方式に基づいて非同期入出力のためのライブラリを用意し，コマンドラインとして“agrep 0222222 ./DATA.2M/0*”を用いた．

3.3.1 ソースコードの書換え

ソースコードの書換えは，open，close，read 関数をそれぞれ hpcsaio_open，hpcsaio_close，hpcsaio_read 関数に置換する．また，対象ループ実行の前後に hpcsaio_init, hpcsaio_term 関数を挿入する．これらのソースコード変換を自動で行う場合について考えると，対象ループ中の open，read，close の置換，また，対象ループの前後の hpcsaio_init, hpcsaip_term の挿入は比較的自明で，HPCS Toolkit が提供する解析木アクセス用及びソースコード変換用ライブラリを用いて実装できる．複数ファイルに跨ったファイルの非同期 read を行なわない場合は上記の自明な変換のみが必須となる．複数ファイルに跨ったファイルの非同期 read を行う場合は，hpcsaio_hint 関数の挿入が必要となる．この場合，ファイル名の保持の形式がプログラムによって異なるため自明とは言えない．今回は，ループ中のファイル open でファイルが配列として指定されていたので hpcsaio_hint API でファイル名の配列をそのまま引数としてライブラリを呼び出した．一方，ループ中での open でのファイル名の指定が配列ではない場合は，対象ループ(“for (i = 0; I < Numfiles; i++)”)をループの前に複製して，複製したループ中に hpcsapi_hint_one を入れて一つずつファイルを指定することで，多くのパターンに対応可能であると考えている．

以下，agrep の書換えの例を擬似コードで示す．

```
/* loop for the specified target files */
for (i = 0; i < Numfiles; i++) {
    /* open a target file */
    if ((fd = open(Textfiles[i], 0)) <= 0) {
        continue;
    }
    /* inlined code of "sgrep" from here */
    while ((num_read =
        read(fd, text+offset, BLOCKSIZE)) > 0)
        {
        /* … search specified pattern
            in the file … */
    }
    /* inlined code of "sgrep" to here */
    close(fd); /* close the target file */
}
```

図 7　最適化前のソースコード(擬似コード)

Figure 7　Pseudo Code before Optimization

```
hpcstaio_init(0, 0);
…
hpcstaio_hint(Numfiles, Textfiles, 0, 0);
…
/* loop for the specified target files */
for (i = 0; i < Numfiles; i++) {
    /* open a target file */
    if ((fd = hpcstaio_open(Textfiles[i], 0)) <=
        0) {
        continue;
    }
    while ((num_read =
        hpcstaio_read(fd,
        text+offset,BLOCKSIZE)) > 0)  {
    /* … search specified pattern in the file … */
    }
    /* close the target file */
    hpcstaio_close(fd);
}
…
hpcstaio_term();
```

図 8　最適化後のソースコード(擬似コード)

Figure 8　Pseudo Code after Optimization

3.3.2 並列度と実行時間

最適化の際の並列度の最適値を確認するために，並列度を 1-16 まで変化させて実行時間を測定した．以下，ファイルシステムに Journal File System (JFS)， 2MB のファイルを 512 個，及び，64KB のファイルを 512 個使用した場合の並列度及び実行時間のグラフである．

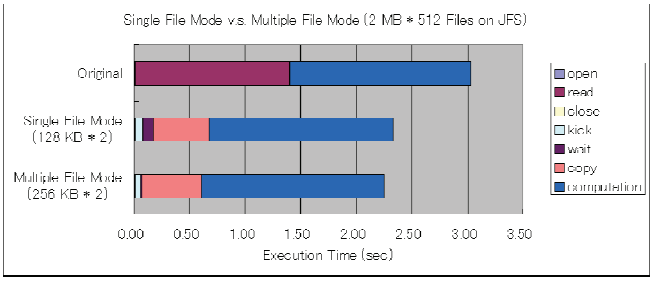
図 9　agrep の並列度と実行時間

Figure 9　Degree of Parallel and Execution Time of Agrep

最良並列度はいずれの場合の 2 となり，19-24%実行時間が削減された．

3.3.3 単一ファイルモードと複数ファイルモード

これまでの結果は，同一ファイル内のみで非同期 read による read 処理の並列化を行ったものであるが，複数ファイルに跨った並列処理を行うことにより，より大きな最適化の効果を得ることが可能となる．以下は，単一ファイル内の read の並列処理を行った場合と複数ファイルに跨った read 処理の並列処理を行った場合のそれぞれ最良のパラメータを用いた場合の測定結果を示す．この例では，複数ファイルに跨った並列処理を行うことにより，単一ファイル内のみの read の並列処理を行った場合と比較して更に約 10%の実行時間の削減が可能となった．



	最短 I/O 時間	I/O 最適化の効果
オリジナル	1.4 秒	-
単一ファイルモード(128KB*2)	0.68 秒	52.0%
複数ファイルモード(256KB*2)	0.61 秒	56.0%

図 10　agrep のファイルの先読み方式と実行時間

Figure 10　Execution Mode and Execution Time of Agrep

4. まとめと今後の課題

blocking read を non-blocking read に変換することにより複数の read 要求を並列に発行し，blocking read の待ち時間を削減することによりファイル I/O の性能を向上させる最適化について検討した．特にこの最適化のための性能解析及びソースコード自動変換による最適化を，HPCS Toolkit のルールとして適用するために必要な各機能について検討した．

また，マイクロベンチマークプログラム及び正規表現検索プログラム agrep を手動で最適化し，最適化前後の性能を測定し，実際に性能が向上することを確認した．正規化表現検索プログラム agrep では read 時間が最大 52%，全体の実行時間が最大 22%向上した．また，その際のソースコード変換方式や性能向上について考察した．

参考文献

1)　High productivity computing systems toolkit.　IBM alphaworks. http://www.alphaworks.ibm.com/tech/hpcst.
2)　G. Cong, I.-H. Chung, H.-F. Wen, D. J. Klepacki, H.Murata, Y.Negishi, and T.Moriyama.　A holistic approach towards automated performance analysis and tuning. In Proceedings　of Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, pages 33-44, Aug. 2009. /
3)　Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur and William Gropp, Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications, Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC), Nov. 2008.
4) S. Akyurek and K. Salem. Adaptive block rearrangement. In Proceedings of the IEEE International Conference on Data Engineering, April 1993.
5) Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP), October 1991.
6) Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. ACM Transaction on Computer Systems (TOCS), 14(4):311–343, 1996.
7) Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In Proceedings of the USENIX Winter 1994 Technical Conference, 1994.
8) K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD), May 1993.
9) Sun Wu, Udi Manber, AGREP - A Fast Approximate Pattern-Matching Tool, Proceedings of the winter 1992 USENIX Conference San Francisco, USA, 20.-24. Jan. 1992,