POLITECHNIKA KRAKOWSKA		
WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ		
Przetwarzanie Równoległe i Rozproszone		
Numer projektu: **1**	Temat ćwiczenia: **Układy równań liniowych metodą Czebyszewa**	**Piotr Ziętek** **Piotr Saletra**
Data wykonania: 05.11.2024	Data oddania: 05.11.2024	Ocena:

1. Wstęp.

1.1 Rozwiązywanie układów równań liniowych metodą Chebysheva (iteracyjną).

Celem niniejszego sprawozdania jest omówienie działania i zastosowania algorytmu rozwiązywania układów równań liniowych przy użyciu metody Czebyszewa, znanej również jako metoda iteracyjna Czebyszewa.

Metoda Czebyszewa - jest to metoda opracowana na podstawie wielomianów Czebyszewa, która pozwala na efektywniejsze przybliżanie rozwiązania w porównaniu do tradycyjnych metod gradientowych. Kluczową zaletą metody Czebyszewa jest jej szybka zbieżność, wynikająca z właściwego doboru parametrów iteracji przy użyciu wartości własnych macierzy AAA. Dzięki temu, metoda Czebyszewa jest szczególnie przydatna w zastosowaniach, gdzie wymagana jest wysoka precyzja obliczeń w stosunkowo krótkim czasie.

W trakcie sprawozdania przeanalizowane zostaną czasy wykonania algorytmu współbieżnego przy użyciu RPC.

1.2 Sun RPC

Sun RPC (**Remote Procedure Call**), która stanowi jedną z podstawowych metod komunikacji między procesami w rozproszonych systemach komputerowych. Sun RPC, opracowane przez firmę Sun Microsystems, umożliwia zdalne wywoływanie procedur, co pozwala na wykonywanie funkcji na zdalnych maszynach w taki sposób, jakby były wywoływane lokalnie. Dzięki temu Sun RPC stanowi kluczowe narzędzie w realizacji usług sieciowych oraz aplikacji działających w architekturze klient-serwer.

2. Listing kodu programu oraz jego działanie

2.1 Kod.

- Plik czebyszew.x

```
/*czebyszew.x */

typedef double DynamicArray[25];

struct LinearSystem {
        int size;
        DynamicArray matrix[25];
        DynamicArray vector;
        double errorTolerance;
        int maxIterations;
};

struct SolverResult {
    DynamicArray solutionVector;
    int iterationsPerformed;
    int status;
};

program CHEBYSHEV_SOLVER_PROG {
    version CHEBYSHEV_SOLVER_VERS {
        SolverResult calculateChebyshevSolution(LinearSystem) = 1;
    } = 1;
} = 0x31234567;
```

Plik chebyszew.x który posłuży nam do wygenerowania stub'a dla serwera oraz klienta (za pomocą polecenia rpcgen). Zawiera on w sobie deklarację tablicy służącej do przechowywania macierzy i wektora wyników. Struktura LinearSystem ma za zadanie "realizować" układ równań liniowych. SolverResult przechowywać będzie wynik. Na samym dole zrzutu widzimy definicję interfejsu RPC z jedną metodą o nazwie calculateChebyshevSolution (przyjmuje ona strukturę LinearSystem, a zwraca SolverResult).

- Plik czebyszew.h

```c
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _CZEBYSZEW_H_RPCGEN
#define _CZEBYSZEW_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


typedef double DynamicArray[25];

struct LinearSystem {
    int size;
    DynamicArray matrix[25];
    DynamicArray vector;
    double errorTolerance;
    int maxIterations;
};
typedef struct LinearSystem LinearSystem;
```

```c
struct SolverResult {
    DynamicArray solutionVector;
    int iterationsPerformed;
    int status;
};
typedef struct SolverResult SolverResult;

#define CHEBYSHEV_SOLVER_PROG 0x31234567
#define CHEBYSHEV_SOLVER_VERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define calculateChebyshevSolution 1
extern  SolverResult * calculatechebyshevsolution_1(LinearSystem *, CLIENT *);
extern  SolverResult * calculatechebyshevsolution_1_svc(LinearSystem *, struct svc_req *);
extern int chebyshev_solver_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define calculateChebyshevSolution 1
extern  SolverResult * calculatechebyshevsolution_1();
extern  SolverResult * calculatechebyshevsolution_1_svc();
extern int chebyshev_solver_prog_1_freeresult ();
#endif /* K&R C */
```

```
/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_DynamicArray (XDR *, DynamicArray);
extern  bool_t xdr_LinearSystem (XDR *, LinearSystem*);
extern  bool_t xdr_SolverResult (XDR *, SolverResult*);

#else /* K&R C */
extern bool_t xdr_DynamicArray ();
extern bool_t xdr_LinearSystem ();
extern bool_t xdr_SolverResult ();

#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_CZEBYSZEW_H_RPCGEN */
```

- Plik czebyszew_xdr.c

```
#include "czebyszew.h"

bool_t
xdr_DynamicArray (XDR *xdrs, DynamicArray objp)
{
    register int32_t *buf;

     if (!xdr_vector (xdrs, (char *)objp, 25,
        sizeof (double), (xdrproc_t) xdr_double))
            return FALSE;
    return TRUE;
}
```

```
bool_t
xdr_LinearSystem (XDR *xdrs, LinearSystem *objp)
{
    register int32_t *buf;

    int i;
     if (!xdr_int (xdrs, &objp->size))
         return FALSE;
     if (!xdr_vector (xdrs, (char *)objp->matrix, 25,
        sizeof (DynamicArray), (xdrproc_t) xdr_DynamicArray))
         return FALSE;
     if (!xdr_DynamicArray (xdrs, objp->vector))
         return FALSE;
     if (!xdr_double (xdrs, &objp->errorTolerance))
         return FALSE;
     if (!xdr_int (xdrs, &objp->maxIterations))
         return FALSE;
    return TRUE;
}
```

```
bool_t
xdr_SolverResult (XDR *xdrs, SolverResult *objp)
{
    register int32_t *buf;

     if (!xdr_DynamicArray (xdrs, objp->solutionVector))
         return FALSE;
     if (!xdr_int (xdrs, &objp->iterationsPerformed))
         return FALSE;
     if (!xdr_int (xdrs, &objp->status))
         return FALSE;
    return TRUE;
}
```

- czebyszew_svc.c

```c
#include "czebyszew.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
```

```c
static void
chebyshev_solver_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        LinearSystem calculatechebyshevsolution_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;

    case calculateChebyshevSolution:
        _xdr_argument = (xdrproc_t) xdr_LinearSystem;
        _xdr_result = (xdrproc_t) xdr_SolverResult;
        local = (char *(*)(char *, struct svc_req *)) calculatechebyshevsolution_1_svc;
        break;

    default:
        svcerr_noproc (transp);
        return;
    }
```

```
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
    }
    return;
}
```

```
int
main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS, chebyshev_solver_prog_1, IPPROTO_UDP)) {
        fprintf (stderr, "%s", "unable to register (CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS, chebyshev_solver_prog_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register (CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS, tcp).");
        exit(1);
    }
```

```
    svc_run ();
    fprintf (stderr, "%s", "svc_run returned");
    exit (1);
    /* NOTREACHED */
}
```

- czebyszew_clnt.c

```c
#include <memory.h> /* for memset */
#include "czebyszew.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

SolverResult *
calculatechebyshevsolution_1(LinearSystem *argp, CLIENT *clnt)
{
    static SolverResult clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, calculateChebyshevSolution,
        (xdrproc_t) xdr_LinearSystem, (caddr_t) argp,
        (xdrproc_t) xdr_SolverResult, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

Powyższe 4 pliki zostają wygenerowane przez plik czebyszew.x. Zgodnie z komentarzem zamieszczonym na górze tych plików nie wprowadzaliśmy w nich żadnych zmian.

- Plik czebyszew_client.c

```c
#include "czebyszew.h"
#include <memory.h>
#include <time.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    CLIENT *client;
    LinearSystem linearSystem;
    SolverResult *solverResult;
    char *serverHost;

    clock_t start, end;
    double cpu_time_used;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <server_host>\n", argv[0]);
        return 1;
    }
    serverHost = argv[1];

    client = clnt_create(serverHost, CHEBYSHEV_SOLVER_PROG, CHEBYSHEV_SOLVER_VERS, "udp");
    if (client == NULL) {
        clnt_pcreateerror(serverHost);
        return 1;
    }
```

```c
linearSystem.size = 3;
linearSystem.matrix[0][0] = 4; linearSystem.matrix[0][1] = 1; linearSystem.matrix[0][2] = 2;
linearSystem.matrix[1][0] = 1; linearSystem.matrix[1][1] = 3; linearSystem.matrix[1][2] = -1;
linearSystem.matrix[2][0] = 2; linearSystem.matrix[2][1] = -1; linearSystem.matrix[2][2] = 3;
linearSystem.vector[0] = 4; linearSystem.vector[1] = 2; linearSystem.vector[2] = 6;
linearSystem.errorTolerance = 0.0001;
linearSystem.maxIterations = 100;

start = clock();

solverResult = calculatechebyshevsolution_1(&linearSystem, client);
if (solverResult == NULL) {
    fprintf(stderr, "Call failed\n");
    clnt_destroy(client);
    return 1;
}

end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Time CPU: %f s\n", cpu_time_used);
```

```c
    if (solverResult->status == 0) {
        printf("Solution found in %d iterations:\n", solverResult->iterationsPerformed);
        for (int i = 0; i < linearSystem.size; i++) {
            printf("x[%d] = %f\n", i, solverResult->solutionVector[i]);
        }
    } else {
        printf("Solution not found within maximum iterations.\n");
    }

    clnt_destroy(client);
    return 0;
}
```

Program łączy się z serwerem przy użyciu protokołu UDP i zdalnie wywołuje funkcję rozwiązującą układ równań. Program oczekuje adresu serwera jako argumentu wywołania. Jeśli nie zostanie podany, program wypisuje instrukcję użycia i kończy działanie. Następnie, funkcja clnt_create nawiązuje połączenie z serwerem o nazwie CHEBYSHEV_SOLVER_PROG na podanym adresie.

Program inicjalizuje przykładowy układ równań liniowych 3x3, tworząc macierz i wektor wyników oraz ustawiając parametry errorTolerance (tolerancję błędu) i maxIterations (maksymalną liczbę iteracji). Następnie, struktura linearSystem jest przesyłana do serwera za pomocą funkcji calculatechebyshevsolution_1, która zwraca strukturę SolverResult zawierającą wyniki obliczeń. W przypadku błędu wywołania zdalnego wypisywany jest odpowiedni komunikat.

- Plik czebyszew_server.c

```c
#include "czebyszew.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
#include <math.h>

#define NUM_THREADS 4

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

SolverResult solverResult;
double x[25] = {0};
double r[25];
pthread_mutex_t lock;
```

```c
typedef struct {
    int start;
    int end;
    double alpha;
    LinearSystem *linearSystem;
} ThreadData;

void* calculateResidual(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    for (int i = data->start; i < data->end; i++) {
        r[i] = data->linearSystem->vector[i];
        for (int j = 0; j < data->linearSystem->size; j++) {
            r[i] -= data->linearSystem->matrix[i][j] * solverResult.solutionVector[j];
        }
    }
    pthread_exit(0);
}

void* updateSolutionVector(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    pthread_mutex_lock(&lock);
    for (int i = data->start; i < data->end; i++) {
        solverResult.solutionVector[i] += data->alpha * r[i];
    }
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}
```

```c
SolverResult *calculatechebyshevsolution_1_svc(LinearSystem *linearSystem, struct svc_req *rqstp)
{
    int size = linearSystem->size;
    double *b = linearSystem->vector;
    double errorTolerance = linearSystem->errorTolerance;
    int maxIterations = linearSystem->maxIterations;

    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];

    solverResult.iterationsPerformed = 0;

    double lambdaMin = 0.612, lambdaMax = 5.576;
    double d = (lambdaMax + lambdaMin) / 2;
    double c = (lambdaMax - lambdaMin) / 2;
```

```
int k;
for (k = 0; k < maxIterations; k++) {
    solverResult.iterationsPerformed = k + 1;

    int chunkSize = size / NUM_THREADS;
    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].linearSystem = linearSystem;
        threadData[i].start = i * chunkSize;
        threadData[i].end = (i == NUM_THREADS - 1) ? size : (i + 1) * chunkSize;
        pthread_create(&threads[i], NULL, calculateResidual, (void*)&threadData[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    double normR = 0;
    for (int i = 0; i < size; i++) {
        normR += r[i] * r[i];
    }
    normR = sqrt(normR);
    if (normR < linearSystem->errorTolerance) {
        solverResult.status = 0;
        return &solverResult;
    }
```

```
        double alpha = 1.0 / (d + c * cos(3.14159265358979323846 * (k + 0.5) / linearSystem->maxIterations));

        for (int i = 0; i < NUM_THREADS; i++) {
            threadData[i].alpha = alpha;
            pthread_create(&threads[i], NULL, updateSolutionVector, (void*)&threadData[i]);
        }
        for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }
    }

    solverResult.status = 1;
    solverResult.iterationsPerformed = maxIterations;
    return &solverResult;
}
```

2.2 Program sekwencyjny.

```
#include "czebyszew.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <math.h>

SolverResult solverResult;
double x[25] = {0};
double r[25];

void calculateResidual(LinearSystem *linearSystem) {
    int size = linearSystem->size;
    for (int i = 0; i < size; i++) {
        r[i] = linearSystem->vector[i];
        for (int j = 0; j < size; j++) {
            r[i] -= linearSystem->matrix[i][j] * solverResult.solutionVector[j];
        }
    }
}
```

```c
void updateSolutionVector(LinearSystem *linearSystem, double alpha) {
    int size = linearSystem->size;
    for (int i = 0; i < size; i++) {
        solverResult.solutionVector[i] += alpha * r[i];
    }
}
```

```c
SolverResult *calculatechebyshevsolution_1_svc(LinearSystem *linearSystem, struct svc_req *rqstp)
{
    int size = linearSystem->size;
    double *b = linearSystem->vector;
    double errorTolerance = linearSystem->errorTolerance;
    int maxIterations = linearSystem->maxIterations;

    solverResult.iterationsPerformed = 0;

    double lambdaMin = 0.612, lambdaMax = 5.576;
    double d = (lambdaMax + lambdaMin) / 2;
    double c = (lambdaMax - lambdaMin) / 2;

    int k;
    for (k = 0; k < maxIterations; k++) {
        solverResult.iterationsPerformed = k + 1;

        calculateResidual(linearSystem);

        double normR = 0;
        for (int i = 0; i < size; i++) {
            normR += r[i] * r[i];
        }
        normR = sqrt(normR);
        if (normR < linearSystem->errorTolerance) {
            solverResult.status = 0;
            return &solverResult;
        }
```

```c
        double alpha = 1.0 / (d + c * cos(3.14159265358979323846 * (k + 0.5) / linearSystem->maxIterations));
        updateSolutionVector(linearSystem, alpha);
    }

    solverResult.status = 1;
    solverResult.iterationsPerformed = maxIterations;
    return &solverResult;
}
```

Program ten działa analogicznie do wersji współbieżnej.

2.3 Porównanie czasów wykonania.

- Czebyszew na wątkach i z wykorzystaniem Sun RPC.

```
Time CPU: 0.000272 s
Solution found in 63 iterations:
x[0] = -1.538390
x[1] = 2.461472
x[2] = 3.846066
```

- Czebyszew sekwencyjny (bez Sun RPC).

```
Time CPU: 0.000081 s
Solution found in 63 iterations:
x[0] = -1.538390
x[1] = 2.461472
x[2] = 3.846066
```

Pomiar czasu wykonania algorytmu Czebyszewa pokazał, że wersja współbieżna z wykorzystaniem RPC wykonała się wolniej niż wersja sekwencyjna bez RPC. Algorytm wykonany współbieżnie i z użyciem RPC osiągnął czas 0.000272 s, podczas gdy wersja sekwencyjna lokalna wykonała się w 0.000081 s. Różnica ta wynika głównie z narzutu związanego z komunikacją sieciową i zarządzaniem wątkami. Współbieżne obliczenia z wykorzystaniem kilku wątków wiążą się z dodatkowymi kosztami synchronizacji, które mogą być znaczące przy mniejszych macierzach lub krótkich operacjach, jak w tym przypadku.

3. Wnioski.

Wyniki przeprowadzonych pomiarów czasu wykonania algorytmu Czebyszewa jasno wskazują na różnice między wersją współbieżną z wykorzystaniem RPC a wersją sekwencyjną bez RPC. Program współbieżny z RPC wykonał się w czasie 0,000272 s, podczas gdy wersja sekwencyjna zrealizowała swoje obliczenia w 0,000081 s. Taki wynik ujawnia, że narzut związany z synchronizacją wątków znacząco wpływa na czas realizacji algorytmu w przypadku podejścia współbieżnego (może też to wynikać z nieoptymalnej implementacji współbieżności). Warto podkreślić, że chociaż podejście współbieżne z użyciem RPC może być korzystne w przypadku bardziej złożonych obliczeń na dużych macierzach, to dla małych problemów obliczeniowych sekwencyjna metoda lokalna okazuje się bardziej wydajna. Ostatecznie, wyniki te podkreślają znaczenie analizy narzutów związanych z komunikacją w kontekście wyboru strategii obliczeniowej w systemach rozproszonych.